Animated 3D Worlds Using Java and SketchUp

Chonmaphat Roonnapak

A Thesis Submitted in Fulfillment of the Requirements for the Degree of Master of Engineering in Computer Engineering

Prince of Songkla University

2015

Thesis Title	Animated 3D Worlds Using Java and SketchUp
Author	Mr. Chonmaphat Roonnapak
Major Program	Computer Engineering

Major Advisor

....
(Dr. Andrew Davison)

Examining Committee :

....Chairperson
(Asst.Prof.Dr. Pichaya Tandayya)

....Committee
(Dr. Andrew Davison)

....Committee
(Dr. Duenpen Kochakornjarupong)

The Graduate School, Prince of Songkla University, has approved this thesis as fulfillment of the requirements for the Master of Engineering Degree in Computer Engineering

....
(Assoc.Prof.Dr.Teerapon Srichana)
Dean of Graduate School

This is to certify that the work here submitted is the result of the candidate’s own investigations. Due acknowledgement has been made of any assistance received.

....Signature
(Dr. Andrew Davison)
Major Advisor

....Signature
(Mr. Chonmaphat Roonnapak)
Candidate

I hereby certify that this work has not been accepted in substance for any degree, and is not being currently submitted in candidature for any degree.

....Signature

(Mr. Chonmaphat Roonnapak)

Candidate

Thesis Title	**Animated 3D Worlds Using Java and SketchUp**
Author	**Mr. Chonmaphat Roonnapak**
Major Program	**Computer Engineering**
Academic Year	**2015**

Abstract

Three-dimensional virtual world is one of those important topics in computer graphics. Its practical applications are extensive and seem to be highly increased. This research introduces a new Java API for producing animated 3D worlds using Java and SketchUp. SketchUp is a useful and popular tool for 3D world creation that provides ease of use and big set of 3D libraries of which Java lacks. The API is composed of several classes: SketchUp initialization, landscape, scenery models, camera, user model, animation. SketchUp initialization handles starting SketchUp and initiating Ruby console. Landscape and scenery are responsible for environment generation including both static and dynamic models e.g. terrain, trees, buildings, and moving cars. User model and camera represent a generation of the user model along with its camera view. Animation handles movement generation and manipulation of each dynamic model in the scene. The animation includes event listener for monitoring and responding to events during the runtime e.g. a collision between models and the end of animations.

SketchUp provides Ruby API which allows 3D models to be generated, imported, and manipulated by executing Ruby scripts instead of manually using drawing tools. The idea of creating the animated 3D scene is to utilize Autoit for inputting Ruby scripts from each Java class to SketchUp's Ruby console in order to generate a visual 3D result of each component in the scene. Then the messages on Ruby console are read back to Java.

ACKNOWLEDGEMENT

I would like to express my sincere appreciation to my thesis supervisor, Dr.Andrew Davison for his priceless help, guidance and constant encouragement throughout the course of this thesis. I am most grateful for his teaching and advice. I wound not have achieved this far and this thesis would not have been completed without all the support that I have received from him.

Gratitude also goes to the committee members, Asst.Prof.Dr.Pichaya Tandayya, and Dr.Duenpen Kochakornjarupong for their profound and carefully aimed comments and suggestions during the oral defense. My gratitude is extended to all my teachers in the Department of Computer Engineering, Prince of Songkla University, for their devoted instruction.

Finally, I most gratefully acknowledge my parents and my friends for all their support throughout the period of this thesis.

Chonmaphat Roonnapak

CONTENT

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

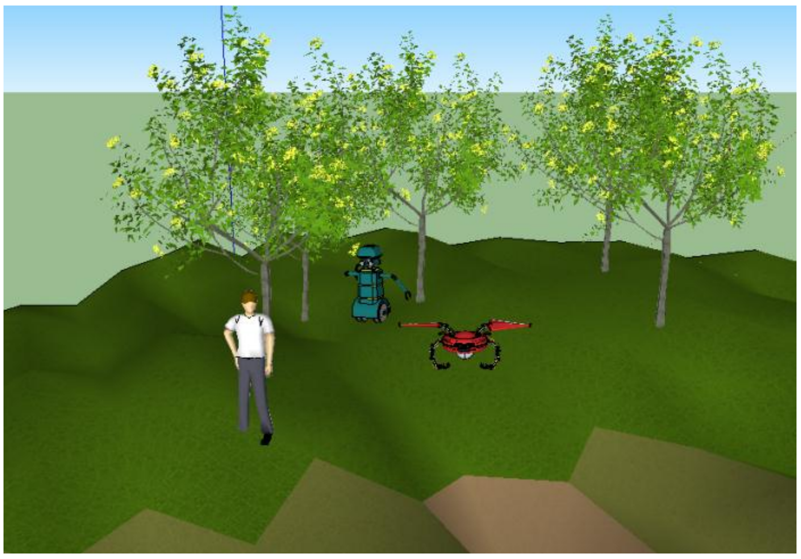
In recent years, 3D virtual world has been one of the most important materials in the field of games and computer graphics. Virtual world is a computer-based simulated environment which interacts to actions and changes. It consists of various 3D elements such as landscape, buildings, vegetation, and avatar representing the user. Its applications can be widely found and the most common form of them is fantasy world in games e.g. Minecraft and Cube World. Both are popular and massive multiplayer online games based on the virtual reality. The virtual worlds are not only limited to games but can also be applied to other fields e.g. engineering, geography, commercial, and entertainment.

There are a number of excellent 3D modeling software that help constructing and rendering the 3D virtual world; 3DS Max [1], Unity 3D [2], AutoCAD [3], IMAGIS [4] and SketchUp [5]. Compared with other modeling software, SketchUp has several advantages such as simplicity, capability, abrupt modeling ability, flexible operation and other free importable online materials [6]. It also provides big set of libraries for 3D world modeling. Not only constructing 3D model, SketchUp also allows the model to be manipulated and animated. SketchUp provides Ruby API which allows users to have a dynamic interpreted open-source programming language called Ruby executed to generate 3D result.

This leads to the idea of this research. An approach for building an animated 3D virtual world with architectural environments using Java and SketchUp is introduced. Compare with Ruby, Java is more widely used, faster and provides more libraries. In Java, there are also functional 3D APIs for 3D virtual world creation e.g. Java3D [7], JOGL [8], and JMonkey Engine [9]. These APIs are well-known and widely employed for 3D modeling in Java, among the sophisticated programmers. However, they were designed to support general needs such as high quality 3D geometric model generation, and game development rather than aiming at animated 3D virtual world generation and some of them are OpenGL-based which makes it difficult for novices to understand and apply.

Since Java is faster than Ruby, it plays an important role in handling calculations in a construction and animation of a 3D scene e.g. calculating coordinates of a terrain and along the path of the animation. Using Java’s Autoit API called jAutoit [10], a Ruby console in SketchUp can be located and Ruby scripts can be delivered to the console and executed in order to generate 3D result of each component of the scene. Then feedback messages on Ruby console are read back to Java for system’s monitoring. It can be concluded that this research creates an interface between Java and 3D modeling tool for creating animated 3D scene.

The API is capable of generating different types of animated 3D scenes. The programmer can customize everything in the scene, for example, the programmer can select whether the terrain is rough or flat and define the number of texture levels. There are several examples that were generated using the implemented API shown in Figure 1.


a) Hill Scene

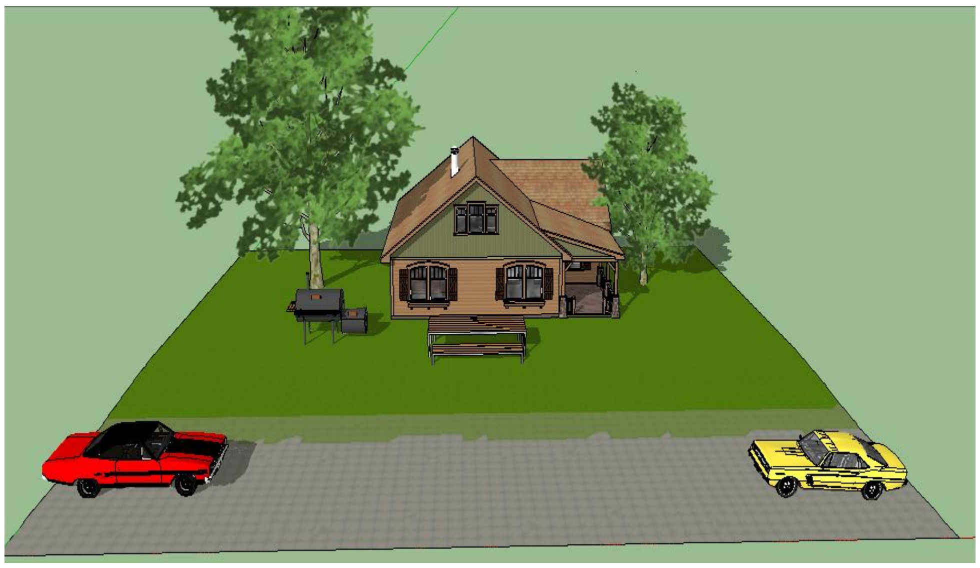

b) House Scene

Figure 1. 3D Scenes Generated from the API

The first scene in Figure 1.a is a grassy hill scene. The scene contains some static trees and dynamic robots positioned on a rough terrain. Another example is a house scene containing static house, trees, griller, table set, and dynamic cars. These models are added on a semi rough terrain generated from a heightmap. The appearance of this type of terrain is up to the heightmap input which makes it flexible for the programmer to design the terrain. The next examples of a city scene in Figure 2 and an apartment scene in Figure 3 represent the use of flat terrain and multiple shapes.

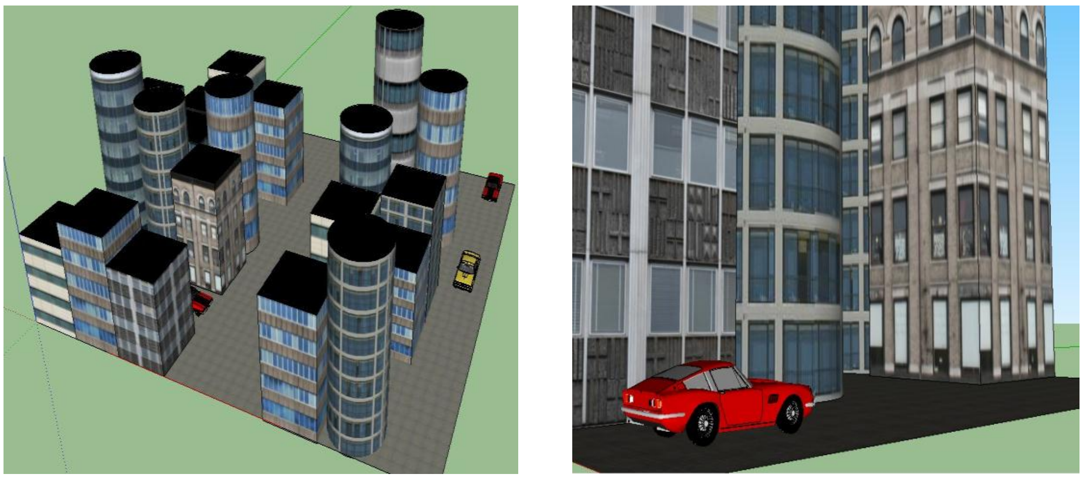


Figure 2. City Scene

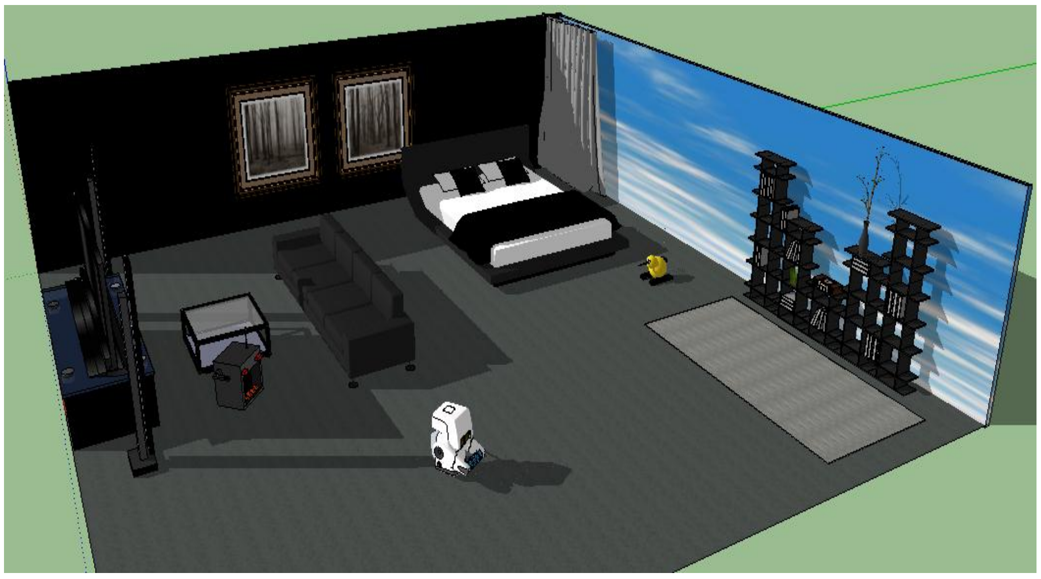


Figure 3. Apartment Room

The result of a city in Figure 2 contains multiple buildings generated from textured blocks and cylinders. The buildings, along with cars, are placed on top of a flat terrain. An apartment scene in Figure 3 also demonstrates a use of flat terrain and shapes as a floor and walls of the room. The room contains dynamic robots and static furniture e.g. sofa, bookshelf, and coffee table. There are a number of scenes that can be produced from the API. However, it depends on the programmer's need, and desire.

The API also supports animation, collision detection and collision recovery. The animation can be generated in the form of a steps array and assigned to each model differently. Each dynamic model will move accordingly to its steps array, after the animation starts. Figure 4 shows an example of animations in a small city scene.

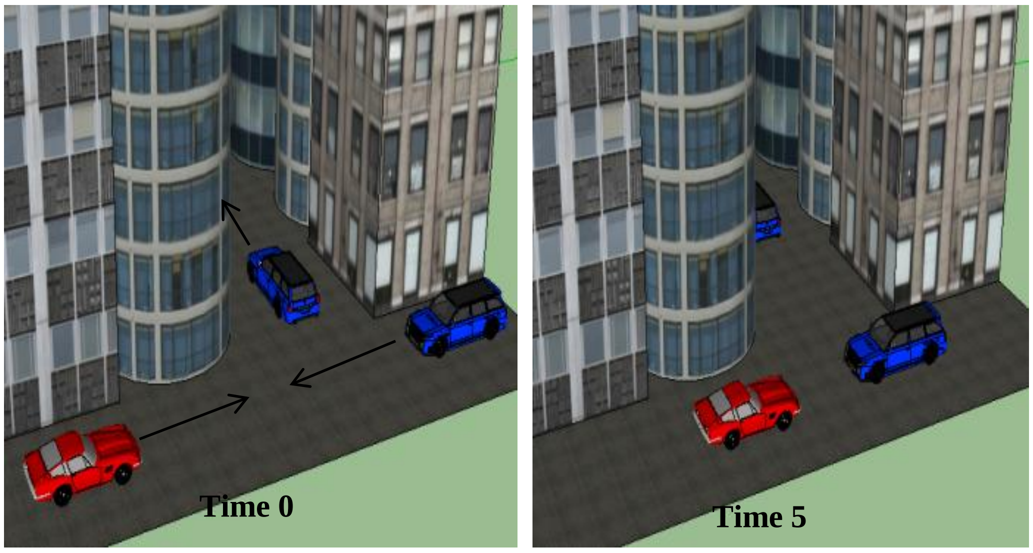


Figure 4. Example of Animation

The animation example in Figure 4 indicates that as the time is incrementing from time zero (at a creation of the scene) to five, each dynamic model is moving continuously for a small distance. Aside from the animation of each dynamic model, the API also supports animation of the camera which is demonstrated in Figure 5.

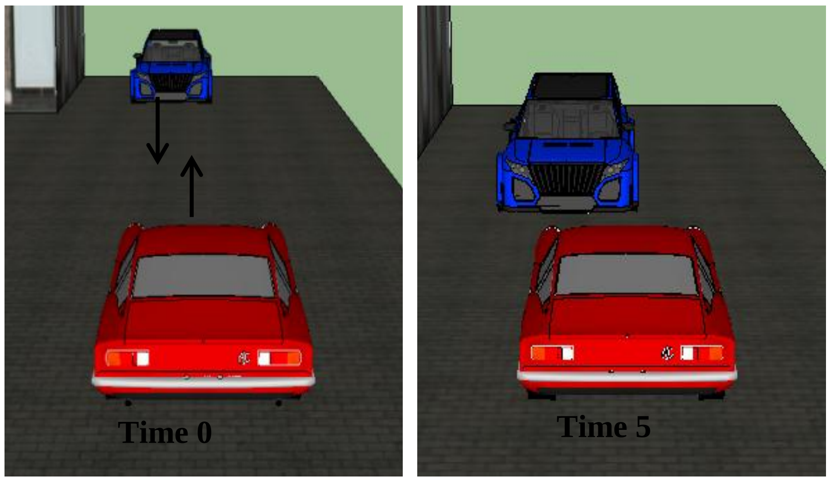


Figure 5. Example of Camera Movement

The camera movement is another feature provided in the API. The programmer can either create a static camera which points to a certain position in the terrain or a dynamic camera which can follow the user model. Figures 5 shows the result after the camera is attached to the user model. The camera follows the model as the model is moving straight during time zero to five.

1.1 Objectives & Scope

Objectives

1. To implement Java API for novice programmers focusing on animated 3D world creation utilizing SketchUp

2. To study SketchUp and implement SketchUp 3D models for landscape, scenery, camera, user model, and discrete-event animation.

3. To study and utilize jAutoit for initiating SketchUp, sending Ruby commands for SketchUp models generation to be executed on SketchUp, and returning feedback messages from SketchUp to Java program

4. To study and implement event listener for monitoring and handling animation-related events such as collision and the end of animations

5. To study implement collision detection and recovery during the animation

Scopes

1. Animated 3D world is generated using Java and SketchUp.

2. The API is able to send Ruby scripts from Java class to Ruby console in SketchUp using jAutoit and retrieve the messages from Ruby console.

3. The use of API will be in high-level so that Java programmer does not have to be familiar with Ruby language.

4. The API contains at least six parts including SketchUp initialization, landscape, scenery, camera view, user model, and animation with collision detection. Each part is defined in the programming level in terms of high-level Java data structures and methods. The next section is the proposed methods and specifications for Java/SketchUp API.

4.1 SketchUp methods

SketchUp methods are responsible for initiating and closing SketchUp and checking for the program's status.

4.2 Landscape methods

The methods are for generating landscape coordinates and creating the 3D landscape by generating a group of polygons using these coordinates. There are three methods for texturing a landscape: initializing texture a number of levels, assigning texture at each level, and applying texture. Texture initializing method defines the name for each texture and maps it with a material. Another method is called in order to apply the defined textures to the landscape surface or objects.

4.3 Scenery methods

There are three activities for scenery: initializing scenery, load scenery and transformation. The methods for initializing the scenery and defining its transformation will be called first in order to create scenery object on Java side. Then a method for scenery loading is called for placing 3D objects onto the terrain on SketchUp side.

4.4 Camera methods

There are two methods for a camera. The first one is the method for setting up the camera to look at a certain position as a static camera. Another method is for attaching to the user model as a dynamic camera.

4.5 Animation methods

There are three methods for animation, the first one is for generating animation for dynamic models, another for applying animation to the object; translation and rotation around Y axis, and the last one is for starting animation and detecting the valid position. The method is called in order to check whether the next position of a model is valid (there is no collision with other objects and the position is within terrain area).

4.6 Listener method

A listener created after the animation starts in order to detect events during the animation, i.e. collision and the end of animation. The listener methods are collision() and finish(). These methods are available for the programmer to define, since the API requires the programmer to deal with each type of event.

1.2 Benefits

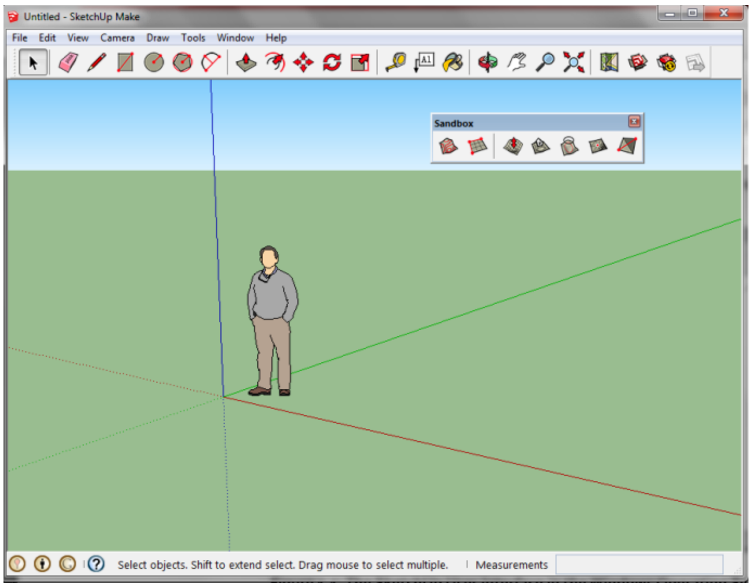
1. To provide a Java programming interface for generating animated 3D world in SketchUp.
2. To provide ease of utilization for novice programmers in order to create 3D worlds
3. To introduce another application of SketchUp which presents simpler modeling ability over other 3D modeling tools

2. RELATED WORKS

This section reviews the popular tools for 3D world creation and existing Java and libraries for 3D scene generation in Ruby and SketchUp. SketchUp, which is the 3D modeling that plays an important role for 3D worlds creation in the API system, is reviewed first, followed by Ruby API in SketchUp, Autoit, and others libraries for 3D scene generation in Java and Ruby including general purpose 3D graphics libraries and 3D gaming libraries.

2.1 Google SketchUp

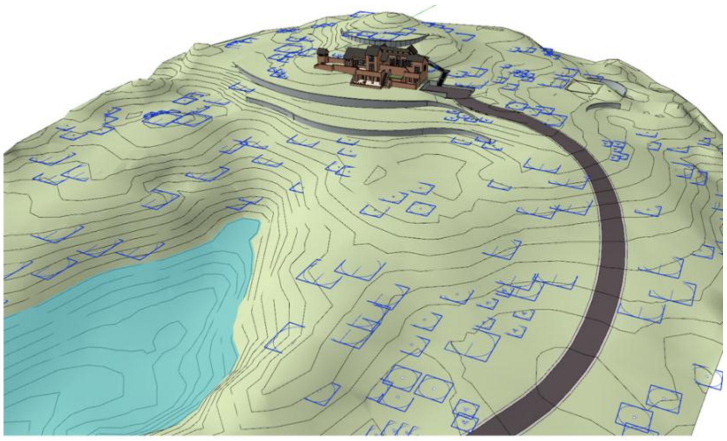
Google SketchUp (SU) is a simple, yet popular, software for 3D modeling and manipulating which covers five areas related to the API needs; landscape, scenery, camera, user model, and animation. As mentioned earlier, it has many advantages that make it outstanding from other 3D modeling software e.g. more extensions, faster modeling ability, smarter toolbars, easier of use, and Ruby API. In this section, the fundamental utilization of SU and the ability for 3D modeling is reviewed.



Select objects. Shift to extend select. Drag mouse to select multiple.

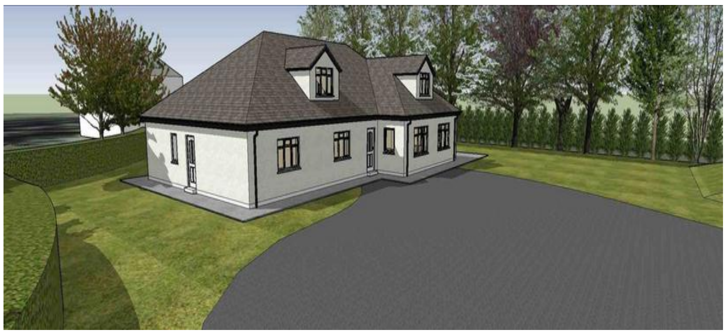
Figure 6. SketchUp interface

Figure 6 shows an interface of SketchUp which is simple but supports a wide range of uses. It contains several tools for 3D modeling which can produce varieties of results started from simple glasses of liquid to realistic grassy hills and a high quality scene in Figure 7.



a) Terrain with buildings [11]

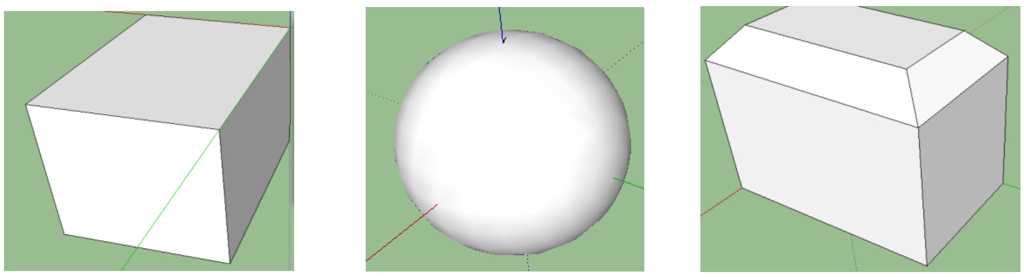
b) Grassy hills created [12]



c) Infrared house model [13]

Figure 7. Examples of SketchUp 3D Scene

SU provides many useful basic drawing tools for users to draw and express out their mental model in forms of visual 2D and 3D graphics. All kinds of object can be created, edited and put together in order to create a bigger and more complex object [14]. Figure 8 displays simple objects drawn by SketchUp tools including a simple box, a sphere, and a complex box.

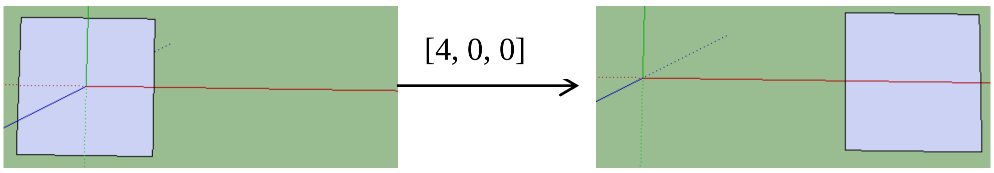


a) Box b) Sphere c) Complex shape

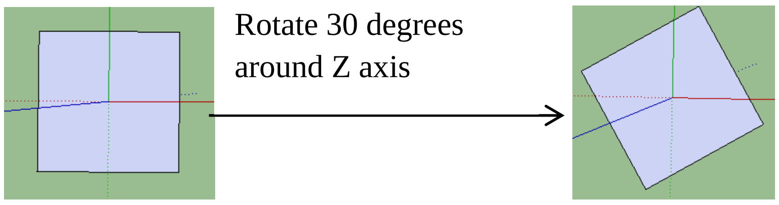
Figure 8. Different 3D Objects Created by SketchUp

Not only creating 3D objects, SU is also capable of manipulating and animating them. It means that all kinds of geometric transformation; translation, scaling and rotation, can be assigned to any objects in order to make their changes and movements. This also counts as one of those benefits of SU, it has the ability to simply manage objects in the scene.

a) Translation,



b) Rotation,



a) Scaling,

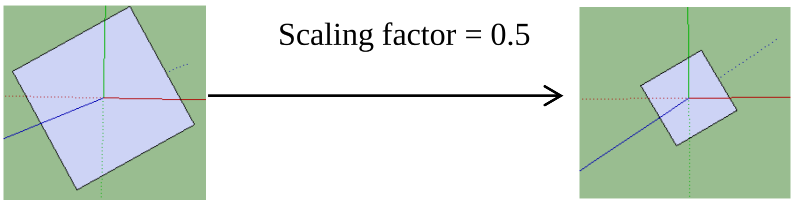


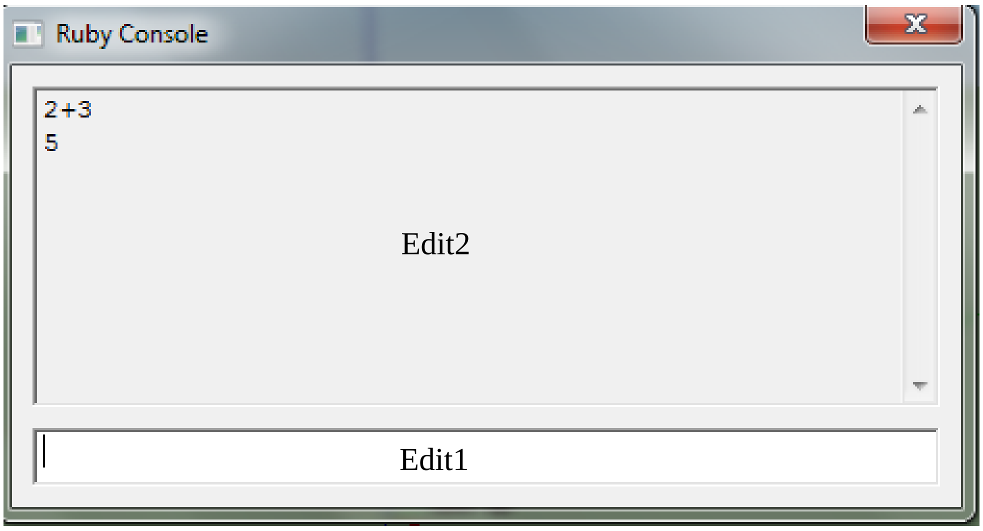
Figure 9. Example of Geometric Transformations

Geometric transformations demonstrating in Figure 9 plays an important role in animation. It allows objects to be moved, rotated and scaled. A complex animation can be created using these basic three kinds of geometric transformation.

2.2 Ruby Console in SketchUp

Ruby is a dynamic, open source programming language with a focus on simplicity and productivity [15]. Ruby is flexible language, it allows user to alter its parts freely. User can basically redefine or remove the essential parts of Ruby. Ruby supports multiple programming paradigms including functional object oriented and imperative. Ruby also provides many useful features such as exception handling, extension libraries and portability. It can be used to implement varieties of programs started from a simple calculator to a complex model of 3D scene. In this research, Ruby is used to perform animated 3D scene creation in SketchUp.

SketchUp provides Ruby API which is the only interface to SketchUp and still the big problem for using SketchUp as an API in other languages. Ruby Console, shown in Figure 10, consists of two parts; Edit1 and Edit2. The console allows the user to type in and execute Ruby commands in Edit1 in order to create, edit, and animate objects in the scene. Then the feedback messages after each execution are displayed in Edit2. Ruby command in SketchUp also supports the need for implementing each part of the API such as initializing Google SketchUp, generating landscape from polygons, creating and positioning dynamic and static scenery, camera setting, and animating objects in the scene including camera view and dynamic models.



```
2+3
5
```

Figure 10. Ruby Console in SketchUp

Executing Ruby script via Ruby Console in SketchUp can also produce the same outcome as using those drawing tools as shown in Figure 11. It is the alternative approach for modeling a 3D scene. Creating objects by script not only can provide the same result as using those tools, but also gives user more ability to manage the objects to be perfectly matched with their mental model, for example, user can define the size and position of the object in a more accurate way.

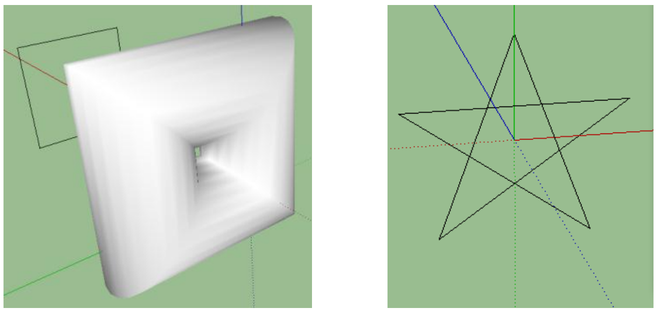


Figure 11. SketchUp Objects Drawn by Ruby Script

The following code is the example of Ruby script for simple object generating and animation which can be executed in Ruby console.

Ruby Code for Box Generation

```
ent = Sketchup.active_model.entities
main_face = ent.add_face [0,0,0], [6,0,0], [6,8,0], [0,8,0]
main_face.pushpull 5
```

Acoording to the code for box generation, In order to generate any object, first the entity in SketchUp which will contain the object has to be declared. To create a box, the square face is required. It can be created using four xyz coordinates (each coordinate is on the same height). After the face is created, it will be pulled up in Z axis in order to construct a box.

Ruby Code for Translating Object

```
ent = Sketchup.active_model.entities
main_face = ent.add_face [0,0,0], [6,0,0], [6,8,0], [0,8,0]
main_face.pushpull 5
tr = Geom::Transformation.translation [0, 0, 5]
ents.transform_entities tr, roof_line
```

After the box is created, the transformation (translation) is declared. The coordinate (0, 0, 5) means the transformation will raise the object up for five units in positive Z axis. Then the transformation is assigned to the box object and elevate it.

Since Java is too general and not suitable to be used for 3D world creation, The idea of combining it with Autoit, Ruby and SketchUp will introduce the new approach for Java to create animated 3D worlds. This will allow Java to be capable of doing other things with SketchUp.

2.3 Autoit

Autoit is a freeware scripting language for automatic administrative tasks. It provides control over window's primary inputs such as mouse and keyboard through a script. Several kinds of tasks can be automatically completed by Autoit e.g. initiating a program, clicking on a close box to terminate an application. It is a small, yet powerful, language which is also capable of handling any complicated actions that can be generalized into a script. The example of Autoit window for writing script is shown in Figure 12.

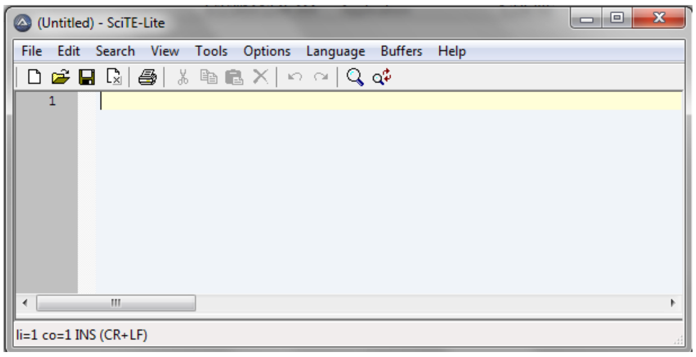


Figure 12. Autoit Window for Writing Script

Autoit has various interesting features. It was designed for ease of utilization, according to its basic syntax which is easy to understand. The automated script can be compiled into an executable version and general user interface (GUI) is also supported. It also provides access to window's DLL and API. Figure 13 shows Autoit Window Info which is a tool for identifying window's title and its components, or even a 2D coordinate representing the position where the mouse cursor is.

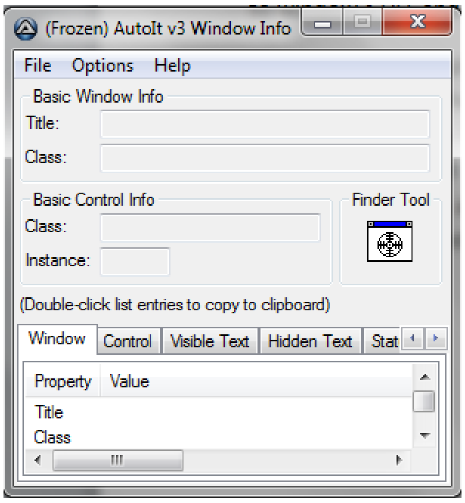
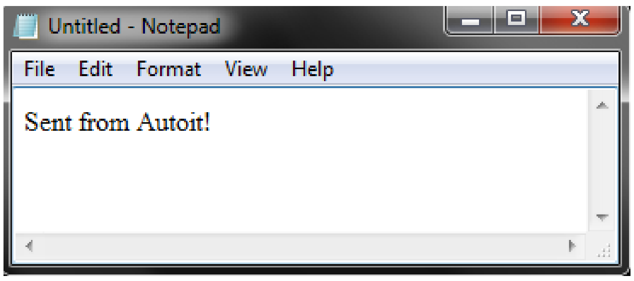

(Frozen) AutoIt v3 Window Info
File Options Help
Basic Window Info
Title:
Class:
Basic Control Info
Class:
Instance:
Finder Tool
(Double-click list entries to copy to clipboard)
Window Control Visible Text Hidden Text Stat
Property Value
Title
Class

Figure 13. Autoit Control Window

The common concept of automating task e.g. writing some texts to Notepad is to initiate the program accordingly to its path, identify the target window and its component where the input will be sent, and then send input to the component based on its name or position on the screen. For example, the following Autoit script launches Notepad from start menu and sends keystrokes to the text field. Figure 14 demonstrates the result after the script is executed.

```
Run("notepad.exe")
WinWaitActive("Untitled - Notepad")
Send("Sent from Autoit!")
```



Sent from Autoit!

Figure 14. Result on Notepad

Autoit also provides an API for Java. This allows Autoit features to be utilized from Java class. Similar steps can be performed by Java class in order to achieve the same result in Figure 14. The following Java code demonstrates the Java commands utilizing Autoit for initiating Notepad and sending texts.

```
private static JAutoIt lib = JAutoIt.INSTANCE;
lib.AU3_Run("notepad.exe", "", 1);
lib.AU3_ControlFocus("Untitled - Notepad", "", "Edit1")
lib.AU3_Send("Sent from Autoit!", 0);
```

The utilization of Autoit features in Java leads to the main concept of this thesis. The approach is to have Java prepares parameters of the 3D virtual worlds and then employs jAutoit to send them along with Ruby commands for virtual world generation to be executed on SketchUp side in order to create visual 3D result.

2.4 General-Purpose 3D Graphics Libraries

The 3D graphic libraries for general needs reviewed in this research are separated into two main categories; OpenGL-based library and scene graph-based library, accordingly to their basic principles of generating 3D graphics.

An OpenGL-based library basically supports OpenGL which is a cross-language, multi-platform application programming interface for 2D and 3D graphics rendering. So this kind of library is available on multiple platforms, yet it can provide more advanced results based on the OpenGL features. On the other hand, a scene graph-based library uses a scene graph as a main principle in order to generate 3D graphics. A scene graph is a general data structure that arranges the logical and spatial representation of a graphical scene. Scene graph helps developers express their mental models and describe the logical relationship between objects in the scene.

2.4.1 OGRE.RB

OGRE.RB is the project that provides a complete and comprehensive wrapper of the OGRE for the Ruby [16]. OGRE (Object-Oriented Graphics Rendering Engine) is a scene-oriented 3D engine designed in order to make it easier to produce an application using hardware-accelerated 3D graphics. The class library provides an interface of the underlying system libraries such as Direct3D and OpenGL based on world object and other intuitive classes [17].

OGRE is simple and easy to use. It provides extensible example framework, flexible mesh data and texture data formats support. The library also supports common requirements, Direct3D and OpenGL, and animation over crossed platforms including Windows, Linux and Mac OSX. Figure 15 displays examples which are created from OGRE.

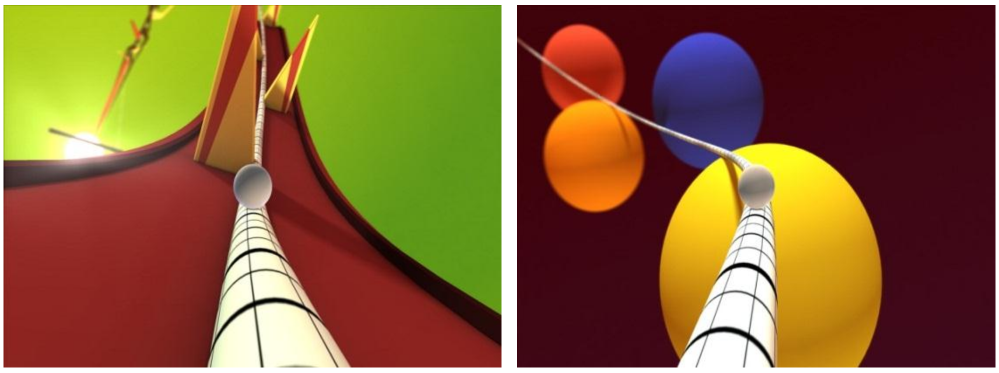


Figure 15. The Screenshots of OGRE Games

OGRE is capable of generating various kinds of result, it can be used to for simulations, business applications and games as shown in Figure 15, but OGRE itself is designed to provide just a world-class graphics solutions. For extra features such as networking, collision, physics and sound, external libraries are needed to be integrated with OGRE, since it is designed to support various requirements of developers.

2.4.2 G3DRuby

G3DRuby is a Ruby extension for G3D library [18]. It is a wrapper classes for many advanced OpenGL features such as vertex arrays and shaders. Basically G3D is used in commercial games, research papers, military simulators, and university courses [19]. It provides a common set of structures which are needed in almost every graphics program. G3DRuby makes low-level libraries like OpenGL easier to be used without limiting functionality. Some of the graphical examples produced from G3DRuby are displayed in Figure 16.

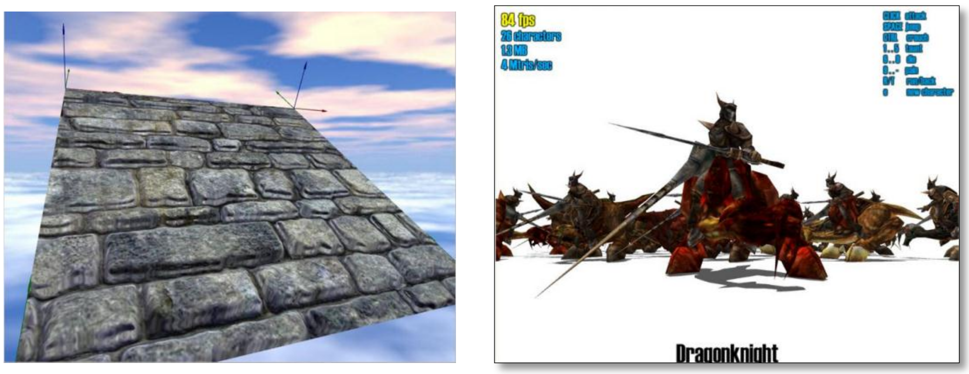


Figure 16. 3D Results Rendered Using G3DRuby

G3DRuby supports multiple rendering modes, hardware-accelerated OpenGL, multiple formats data import and export, and animation. It also provides mesh features, collision detection and textures which make it suitable for creating light-weight games and demos in Ruby.

2.4.3 JOGL

JOGL (Java OpenGL) is a wrapper library allowing Java applications to access the OpenGL API for graphics programming. JOGL allows computer graphics programmers to use the object-oriented tools for Java with hardware-accellerated 2D and 3D graphics [8].

JOGL is different from some other Java OpenGL wrapper libraries. It exposes the procedural OpenGL API via methods on a few classes, not mapping OpenGL functionality onto the object-oriented programming approach. Most of the JOGL method is autogenerated from the OpenGL C header files using a conversion tool. The 3D result from JOGL is shown in Figure 17.

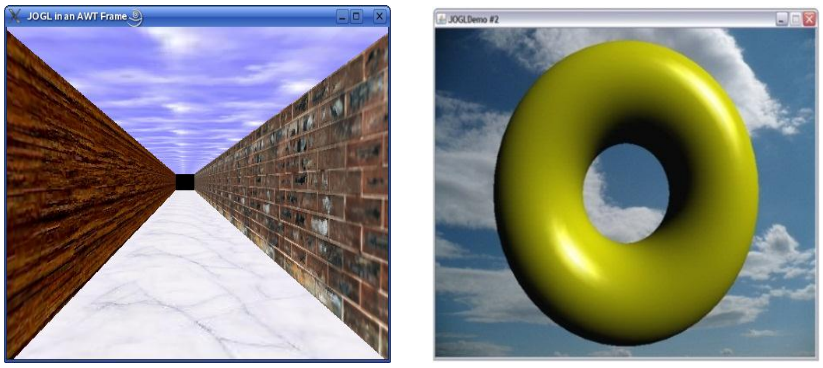


Figure 17. 3D Results of JOGL

JOGL’s main features makes it capable of drawing 3D OpenGL graphics on top of Java2D rendering, overlaying Swing components top of OpenGL rendering, using 3D graphics anywhere where ordinarily a Swing widget would be used, and drawing Java2D graphics on top of 3D OpenGL rendering.

2.4.4 OpenSceneGraph

The OpenSceneGraph is an open source and high performance 3D graphics toolkit. It is cross-platform and used by application developers in visual simulation, games, virtual reality, scientific visualization and modeling fields. It provides high-level rendering features not found in the OpenGL API. OpenSceneGraph also supports application development for mobile platforms such as ios and Android [20].

OpenSceneGraph is written in standard C++, but it also has the binding for Ruby programming language. The software uses the scene graph approach for representing 3D worlds as a graph of node. OpenSceneGraph supports OpenGL and this makes it possible to support from old hardware and operating system to the latest devices. Exceptional results from OpenSceneGraph are shown in Figure 18.

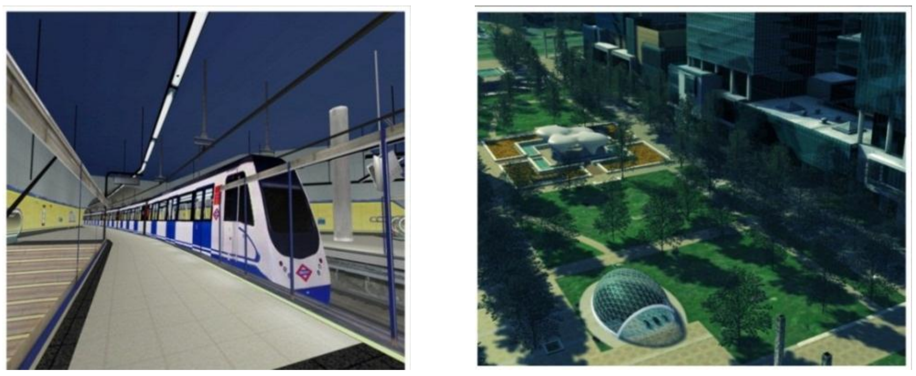


Figure 18. 3D Scene Rendered by OpenSceneGraph

OpenSceneGraph is portable, scalable, and cross-platform. The library also supports multi-language, spatial organization, large database for creating geospatial terrain, and texture mapping. The examples in Figure 18 prove that OpenSceneGraph is capable of generating high quality 3D virtual worlds.

2.4.5 Java3D

The Java 3D is a 3D high-level graphics application programming interface (API) that can be used to create high performance 3D graphics applications in Java platform. It supports both OpenGL and Direct3D. Java 3D is a scene graph based API that encapsulates the graphics programming using a true object-oriented approach. A scene is constructed using a scene graph representing the objects that will be created in the scene. Java 3D is capable of generating various types of result, form a simple object such as a car to more complicated object with multi-texturing, according to Figure 19.

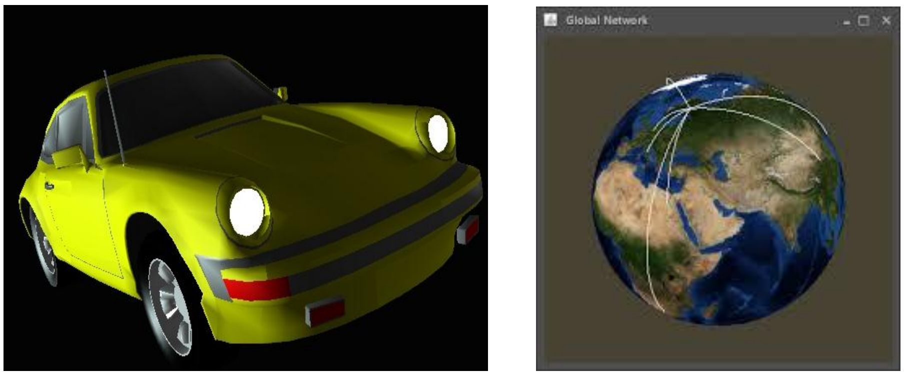


Figure 19. The 3D Objects Generated by Java3D

Java3D provides many useful features such as multithreaded scene graph structure, 3D spatial sound. It is cross-platform and real-time API for both gaming and visualization. The library also supports hardware-accelerated JOGL, OpenGL, and Direct3D renderer which makes it complicated to use for beginners.

2.4.6 Summary

Both OpenGL-based library and scene graph-based library are designed in order to support general and various needs of developers. Both are basically based on OpenGL, so the 3D graphics provided by this type of library is considered acceptable and even exceptional in some engines. However, based on the fact that OpenGL is complex 3D graphic API, using these libraries to generate decent 3D scenes will strongly requires sophisticated developers. Due to its difficulty and complexity, no matter how good and popular it is, OpenGL is too complex for an expected animated 3D scene in this research. For example, to generate a textured surface, vertex shader has to be used as a texturing unit and it is too complicated for a simple animated 3D scene generation.

Another point to be considered is that these libraries are designed mostly to support general needs rather than aiming specifically at animated 3D scene generation, so it can be concluded that these libraries are too complicated, extensive and general for an animated 3D scene generation. It might be more efficient to look for more suitable tool.

2.5 3D Gaming Libraries

3D gaming library is designed based on 3D games development. Most of them are high-performance and capable of rendering rich-detailed 3D graphics. In this section, Irrlicht and jMonkey, the advanced 3D game development engine, will be reviewed.

2.5.1 Irrlicht Engine

The Irrlicht Engine is an open source high performance realtime 3D engine written in C++ but also has an interface for Ruby called Irrlicht/Ruby Interface [21]. It is cross-platform, officially running on Windows, Mac OS X and Linux. Irrlicht supports D3D, OpenGL and also provides its own software renderers [22]. Moreover, there are lots of projects that are developed using Irrlicht engine.

Irrlicht is known for its small size, compatibility with any hardware and a large friendly community which enhancements such as terrain renderers, world layers and tutorials are provided. Yet, it can handle all state of features, which can be found in 3D engines, and many kinds of file formats. The example 3D game projects that are developed based on Irrlicht engine is shown in Figure 20.

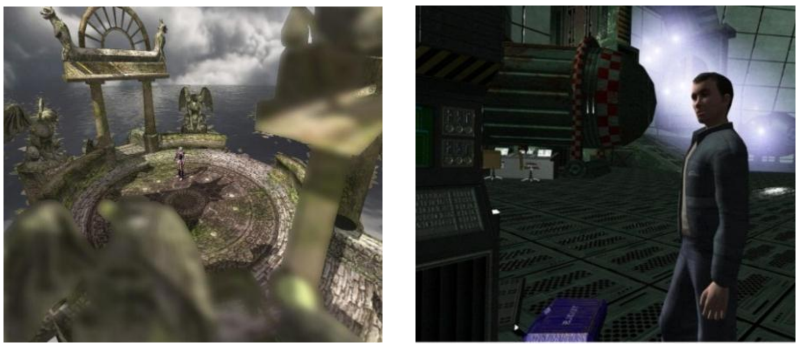
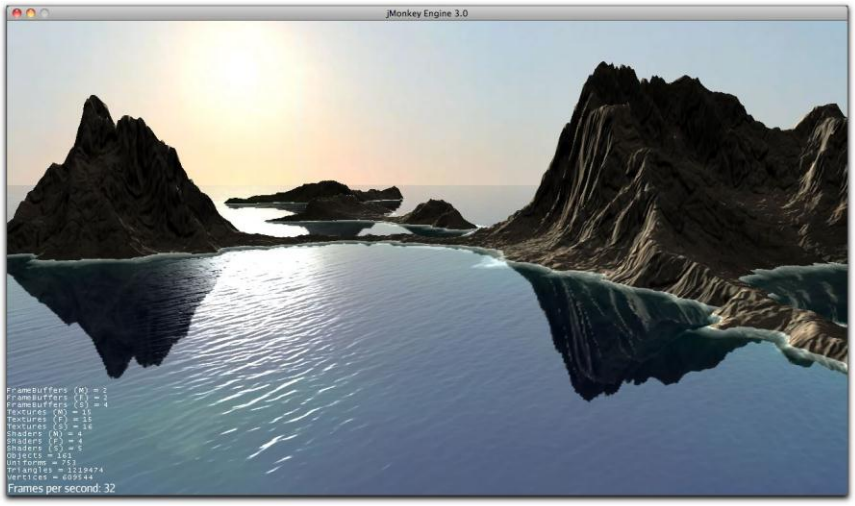


Figure 20. Game Projects Developed Using Irrlicht Engine

Irrlicht engine is powerful, customizeable, platform independent, and easy to use. It supports direct import of common mesh and texture file formats, collision detection, and character animation system with skeletal animation, and high performance real-time 3D rendering using Direct3D and OpenGL.

2.5.2 JMonkey Engine

jMonkeyEngine (jME) is a game engine designed for modern 3D game development [9]. It's a free, open source game engine for *Java game developers* who want to create 3D games using modern technology. jME is programmed entirely in Java due to wide accessibility and quick deployment. jMonkeyEngine is a collection of libraries which makes it a low-level game development tool that doesn't support the general need of visual RPG Maker. However, if users are familiar with programming and OpenGL, they will get the most out of the engine since it fully supports OpenGL 2 through OpenGL 4. JMonkey Engine provides many types of lighting, physical simulation, and special effects. It also supports multiple plat forms and formats of file, terrain generation, and multithreading. The visual examples of JMonkeyEngine are shown in Figure 21.



```
FrameBuffers (M) = 2
FrameBuffers (F) = 2
FrameBuffers (S) = 4
Textures (M) = 15
Textures (F) = 15
Textures (S) = 16
Shaders (M) = 4
Shaders (F) = 4
Shaders (S) = 5
Objects = 161
Uniforms = 753
Triangles = 1219474
Vertices = 609544
Frames per second: 32
```

Figure 21. 3D Scene Generated Using jMonkey Engine

According to Figure 21, the result is beautifully generated using jMonkey Engine, since it is based on OpenGL. However, there is still a complication in implementing collision detection using JMonkeyEngine. The API requires some studies of its classes e.g. RigidBodyControl in order to implement physics and create a collidable scene which also turns out to be many lines of code.

2.5.3 Summary

3D Gaming Library is interesting and perfectly matches the needs of most 3D game developers since it focuses on terrain generation, models, texturing, animation, collision detection and physics. Besides, the quality of 3D graphics rendered by the library is also exceptional. These advantages also make the library better suited with the need at animated 3D scene generation than the previous libraries.

Because the goal is to generate a simple animated 3D scene without too many details, high-leveled graphics and complexity expected, the libraries turn out not to be the best candidates, compared to SketchUp. SketchUp is designed aiming directly at 3D modeling of buildings and landscape. SketchUp is also a good tool for model measurement, printing and exporting. The huge advantage of SketchUp over other full-featured graphics and game library is simplicity. Users can easily express their mental model out using SketchUp which also provides faster rendering speed and requires no OpenGL basics. Moreover, SketchUp has a huge community where tutorials, plugins and extensions are shared, for example, SketchUp’s 3D Warehouse, a vast component library, allows user to download tons of components from others or even upload their own. Yet, SketchUp can be integrated with other Google software such as Google Earth and Google Map, along with its street-view features. It can be concluded that SketchUp is the best tools to be used for developing a simple animated 3D scene.

3. DESIGN

The 3D world generating system consists of two parts, according to Figure 22. The first one is a set of Java classes for characterizing each part of the 3D scene, e.g. the appearance of terrain and scenery models. Another is the result rendering unit handled by SketchUp which converts an input of the animated 3D world described in the Java classes to the visual form of output. This chapter explains the overview of each module of the API and discusses problems of the similar module in the existing 3D API.

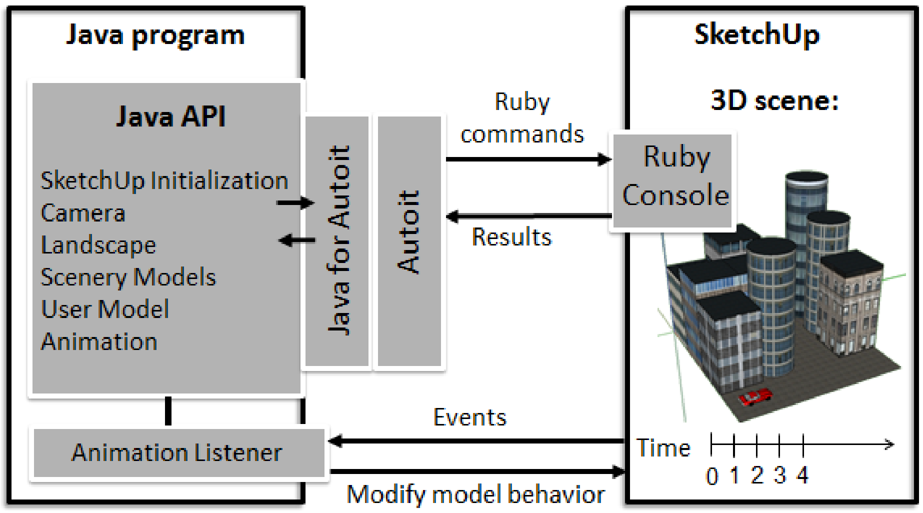


Figure 22. 3D World Generating System Overview

The main module for creating animated 3D scene is Java program utilizing the API for computing, constructing, and manipulating 3D scene. The API includes classes for SketchUp initialization, landscape, scenery, camera, user model, and animation, as shown in Appendix B.1. Each class characterizes properties and modifies status of each component of 3D scene. SketchUp initialization, in Appendix B3, is responsible for initiating and closing SketchUp and Ruby console, to which Ruby command will be sent. It utilizes Java interface for Autoit to automatically opening and closing SketchUp and Ruby console accordingly to the path specified by the programmer.

Other classes control environments in the 3D scene generated in SketchUp. Landscape class, from Appendix B.4, contains parameters and methods for terrain generation that can be customized by the programmer. The programmer can manipulate the terrain's properties provided by the class e.g. altering terrain size, choosing terrain textures and selecting between rough and flat terrain. Scenery model consists of model properties and model generation methods. The programmer can create both dynamic and static models and specify their properties, as described in

Appendix B.2, such as model name, model file, position, and orientation before sending Ruby command to SketchUp in order to start loading the model from file and positioning it relatively to the defined properties. In the 3D scene, there is a camera view controlled by the camera method in Landscape class. The programmer can position the camera and define its eye and target. Another class is User which represents the user model of the scene. The user model can also be followed by the camera so that it can explorer the scene. The last class is Animation. The class, from Appendix B.9, generates movements of each dynamic model in the scene, including the user model, starts the animation and monitors for events during the animation using AnimationListener class in Appendix B13. The AnimationListener is a Java thread created after the animation starts for monitoring for changes in the project directory such as file creation and modification. Once the events occur, such as collision between models and the end of animation, SketchUp will create a new file in the project directory and write the information of the event to the file. Then AnimationListener, after detecting the event, reads the information in the file, create event object, and call a method for responding to each event. The API allows the programmer to define these responding methods, however, a method for simply dealing with a collision is also provided.

The middle component utilizes jAutoIt, an AutoIt API for Java explained before in Chapter two, for arranging communications between Java classes and SketchUp. It delivers Ruby commands, along with values of class's constants, from Java to Ruby console in SketchUp for generating 3D result out of each part of the scene. After each command is executed, there will be messages on the console, jAutoit then reads these feedbacks and reports back to the Java program. The last component is result-rendering unit handled by SketchUp. SketchUp executes Ruby commands, sent from Java, and converts an input of animated 3D world defined by each Java class to the visual 3D output.

3.1 SketchUp Initialization

The main aim of the API design is to create a communication between Java SketchUp so that Java can send series of Ruby commands for 3D world creation to be executed on SketchUp side. This requires SketchUp to be activated first before the commands are sent. The API allows the programmer to automatically initiate SketchUp and its Ruby console as shown in Figure 23 by executing a few line of code.

```
SketchUp skp = new SketchUp();
skp.startSketchUp();
```

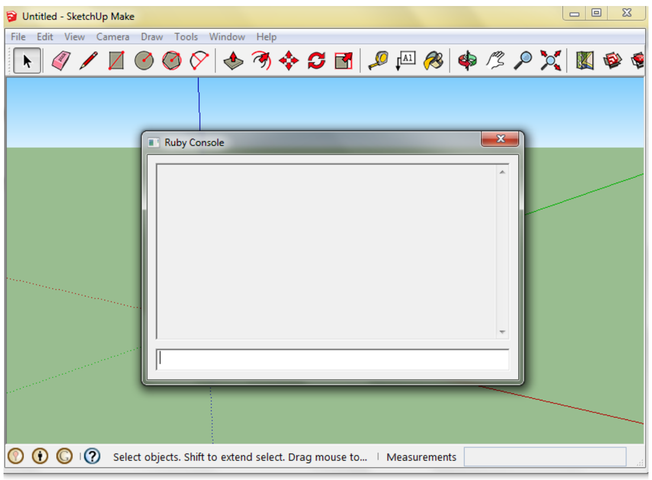

Untitled - SketchUp Make
File Edit View Camera Draw Tools Window Help
Ruby Console
Select objects. Shift to extend select. Drag mouse to...
Measurements

Figure 23. New SketchUp Window

The result from the line of code above is a new SketchUp window, along with an active Ruby console which is ready to receive the commands as displayed in Figure 23. The similar process of initiating a new window or a new project in the existing APIs is more complicated. The following C++ code [23] is a part of an initialization of a new window in OpenGL, a popular 3D programming interface that is supported by a large number of 3D APIs nowadays.

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevious, LPSTR
lpCmdString, int CmdShow)
{
   WNDCLASS wc;
   MSG msg;
   wc.cbClsExtra = 0;
   wc.cbWndExtra = 0;
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
   wc.hCursor = LoadCursor (NULL, IDC_ARROW);
   wc.hIcon = LoadIcon (NULL, IDI_APPLICATION);
   wc.hInstance = hInstance;
   wc.lpfnWndProc = WndProc;
   wc.lpszClassName = "ME";
   wc.lpszMenuName = NULL;
   wc.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
   if (!RegisterClass(&wc))
   {
      MessageBox (NULL,"Error: Cannot Register Class", "ERROR!",
      MB_OK);
      return (0);
   }
   Window = CreateWindow("ME", "Map Editor", WS_OVERLAPPEDWINDOW |
   WS_VISIBLE, 0, 0, 640, 480, NULL, NULL, hInstance, NULL);
}
```

It can be noticed that there are many lines of code which requires some understanding in order to define the window parameters correctly, e.g. background color, cursor, and style, and start creating the window. Despite the complexity of the code, OpenGL still provides general and flexible options for sophisticated programmers to initiate the window. However, since our goal is to implement Java's 3D-world-generating API which is possible and easily understandable for the programmers who are new 3D, less generality and more simplicity of each step throughout the process of generating the world, including the initiation stage, needs to be emphasized. This is the main reason why the API introduces a simpler approach of initializing the new project window in a single line.

3.2 Camera

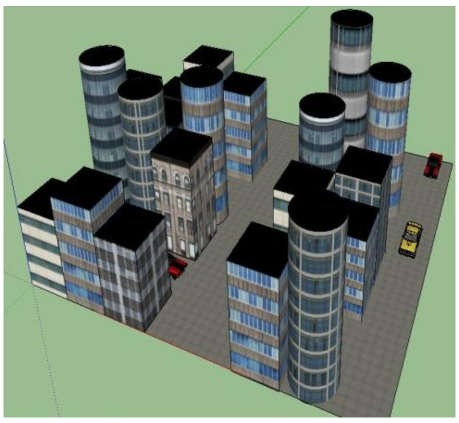
The API allows two types of camera view: static camera and dynamic camera. The static camera in Figure 24.a is represented by 3D eye and target coordinates. These type of camera stares at a certain position and cannot be animated at all during the animation. The dynamic camera in Figure 24.b can be assigned to follow the user model as it is animating. The following code is the usage of the API's method in order to initialize both types of cameras.

Static Camera

```
Point3d eye = new Point3d(151, 164, 168);
Point3d target = new Point3d(94, 705, 400);
terrain.setCam(eye, target);
```

Dynamic Camera

```
Point3d eye = new Point3d(120, 170, 50);
user.setCam(eye);
```


a) Static Camera

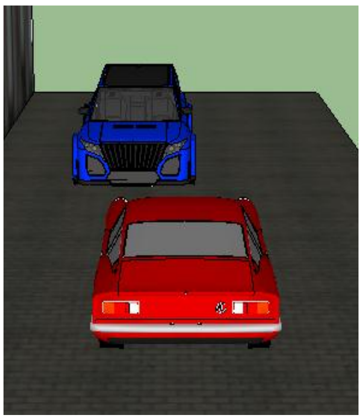

b) Dynamic Camera

Figure 24. Static and Dynamic Cameras

Camera is one of the most important features in any 3D libraries. Some libraries, especially OpenGL-based ones, introduce a complex approaches for setting up the camera, as well as animating it. These camera set-up methods are understandable by the programmers who are familiar with OpenGL and 3D APIs, but time-consumed for those who are new to 3D. The following C++ code [24] example demonstrates the OpenGL approach for preparing a camera's parameters.

```
glm::vec3 cameraPos = glm::vec3(151.0f, 164.0f, 168.0f);
glm::vec3 cameraTarget = glm::vec3(94.0f, 705.0f, 400.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos -
                                            cameraTarget);
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up,
                            cameraDirection));
glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```

During the process of setting up the camera, OpenGL requires the programmer to define a new 3D vector space for the camera, along with the position of the camera itself. This may not involve complex calculation, but it would be easier for the novices, if this can be simplified and converted into couple lines of code.

3.3 Landscape

Landscape is considered the main static object in the 3D scene. All of the models in the scene are constructed on the surface of the landscape. The API allows the landscape to be generated using Diamond-Square algorithm and from 2D heightmap. The existing Diamond-Square algorithm [25] provides a random terrain result, while the terrain generation from heightmap creates a terrain accordingly to the pixel color in the map and provides more certain landscape result. More mechanisms of both approaches will be explained in more detail in later chapter five. The code for terrain generation in the API is simple, yet able to provide complex terrain results. In order to create a terrain, the programmer only needs to specify a few parameters, e.g. terrain depth, resolution, and polygon size.

Terrain Generation from Diamond-Square algorithm

```
public Terrain(int depth, int resolution, int plgSize,)
```

Terrain Generation from Heightmap

```
public Terrain(String file, int depth, int plgSize)
```

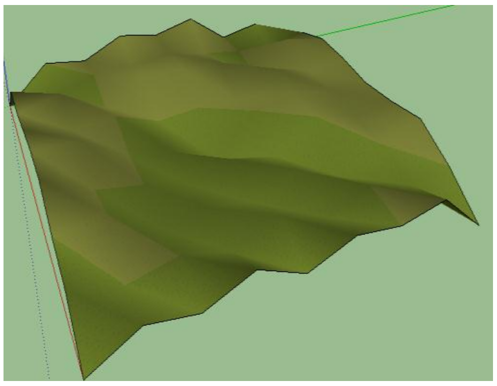
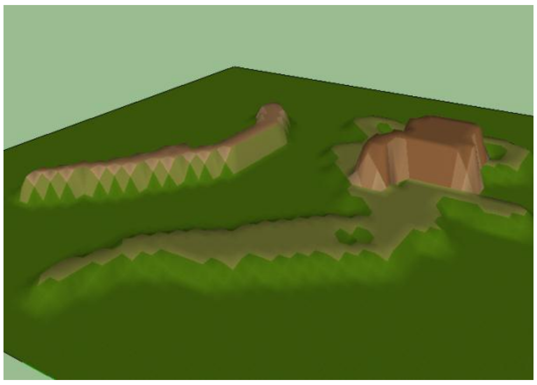



Figure 25. Terrain Results

Figure 25 presents the results of terrain models from both Diamond-Square algorithm at the left and heightmap at the right. Terrain generation is one of the most popular topics of 3D and can be handled by most of the 3D APIs nowadays. The following of Java code [26] demonstrates the terrain generation from heightmap in JOGL, the OpenGL based API in Java.

```
private void renderHeightMap(GL gl, byte[] pHeightMap)
{
 if(renderType == RenderType.LINE)
   gl.glBegin(gl.GL_LINES);
 else
   gl.glBegin(gl.GL_QUADS);

 for (int X = 0; X < (MAP_SIZE - STEP_SIZE); X += STEP_SIZE)
   for (int Y = 0; Y < (MAP_SIZE - STEP_SIZE); Y += STEP_SIZE)
   {
     // downleft vertex
     int x = X;
     int y = height(pHeightMap, X, Y);
    int z = Y;
    if(renderType == RenderType.TEXTURED)
      gl.glTexCoord2f((float)x / (float)MAP_SIZE, (float)z /
      (float)MAP_SIZE);
    else
      setVertexColor(gl, pHeightMap, x, z);
      gl.glVertex3i(x, y, z);

    // upleft vertex
    x = X;
    y = height(pHeightMap, X, Y + STEP_SIZE);
    z = Y + STEP_SIZE;
    if(renderType == RenderType.TEXTURED)
      gl.glTexCoord2f((float)x / (float)MAP_SIZE, (float)(z + 1) /
      (float)MAP_SIZE);
    else
      setVertexColor(gl, pHeightMap, x, z);
      gl.glVertex3i(x, y, z);
    // code for upright vertex
    // code for downright vertex
    }
  gl.glEnd();
  gl.glColor4f(1.0f, 1.0f, 1.0f, 1.0f); // reset
}
```

The code shows a loop for processing each pixel of the heightmap in positive x and y axes direction, calculate a height of the coordinate from the pixel color, and extracting the texture coordinate for later texture mapping. The process of terrain generation by heightmap in JOGL seems to require knowledge about image file processing and also texture mapping, which may not be the perfect option for the programmers who are new to 3D and it would be better if this can be performed by fewer lines of code.

Lighting and shadow is another feature which improves realism of the landscape, along with other models in the scene. It creates light source of the scene and projects shadow of each model on the terrain surface. Lighting in our API is simple to activate by a single line of Java code.

```
  terrain.enableShadow();
```

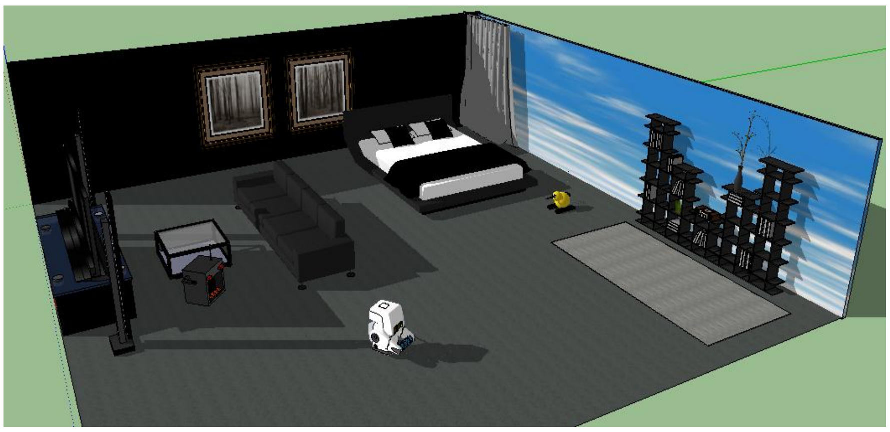


Figure 26. Apartment Room after Enabling Shadow

The result after lighting and shadow in Figure 26 proves that the overall appearance of the scene is improved after shadow is added. More complication in enabling lighting and shadow is found in other OpenGL-based APIs, for example, the programmer is required to manually set up the light source and type of material in the scene as shown in the example code [27].

```
   // Bright white light - full intensity RGB values
   GLfloat ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };
   // Enable lighting
   glEnable(GL_LIGHTING);
   // Set light model to use ambient light
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);
```

The programmer needs to understand each type of the light source available in OpenGL, including specular, diffuse, and ambient light, before creating the light source. Full intensity white ambient light is used in this example as a light source.

Then, the light is enabled by method glEnable(), and the light model specified in ambientLight can be set. However, this is still not the end of the process. The programmer also needs to define the material properties of the polygons in the scene so that they reflect the light.

```
Glfloat gray[] = { 0.75f, 0.75f, 0.75f, 1.0f };
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
glBegin(GL_TRIANGLES);
  glVertex3f(-15.0f,0.0f,30.0f);
  glVertex3f(0.0f, 15.0f, 30.0f);
  glVertex3f(0.0f, 0.0f, -56.0f);
glEnd();
```

The code shows a method for setting the material properties which allows the programmers to define the RGBA value of each material property at the front and the back of a polygon. Then, the polygon which reflects to the ambient light source can be drawn. The whole process of enabling lighting in the OpenGL-based API involves several steps which might be confusing for novice programmers and our API introduces a simpler way which can provide as good result.

3.4 Scenery

Scenery in the 3D world is a set of both 3D models which presents the location of the scene. The API allows the scenery to be drawn in the form of shapes, e.g. towers and buildings, or imported from the existing 3D model files supported by SketchUp, e.g. 3D car models and robot. Scenery can be categorized into two types; static and dynamic. Static sceneries are the models that are unable to move, e.g. trees, tower, and buildings, while dynamic sceneries are those which are assigned by sequences of movements and able to move once the animation starts. The following Java code demonstrates how scenery is created by importing existing SketchUp model using the API.

```
Model tree = new Model(
               "palmTrees.skp", //SketchUp model file name
               new Point2f(100,100), //position
               0, //offset in Z axis
               1, //scale
               0, //rotation
               false //not dynamic model
             );

terrain.addModel(tree);
```

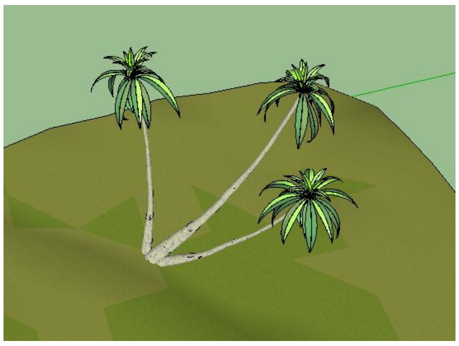


Figure 27. Static Trees

In order to import the tree model as shown in Figure 27, the programmers only need to create the model instance by specifying model file name, position in 2D dimension, offset in Z axis, rotation, and scale of the model. Then, the model can be added to the scene by calling Terrain's method addModel().

The existing OpenGL-based APIs also allow the 3D model to be imported as well, but the process is more complicated. For example, in order to import the model from 3ds Max, the model needs to be exported as ASCII file (.ASE format) containing texts defining the model's properties which are understandable by OpenGL, e.g. GEOMOBJECT, MESH_FACE, and MESH_TFACE. Then, the information in the file is loaded and parsed to OpenGL which involves file processing and memory allocation. Once the file is read, the process of drawing the model can begin. The following code shows only part of C++ code [23] for extracting model information from the ASCII file.

```
if (strcmp(tmp, KEYWORD_MATERIAL_COUNT) == 0)
{
   max_materials = strtol(strtok(NULL,token), NULL, 10);
   mdl_material = new MODEL_MATERIAL[max_materials+1];
}
else if (strcmp(tmp, KEYWORD_BITMAP) == 0)
{
   cur_material++;
   strcpy (mdl_material[cur_material].filename, strtok (NULL,
   token));
}
else if (strcmp(tmp, KEYWORD_OBJECT) == 0)
{
   cur_object++;
   mdl_object[cur_object].max_vertices = 0;
   mdl_object[cur_object].max_triangles = 0;
   mdl_object[cur_object].max_uv_coord = 0;
   mdl_object[cur_object].vertex = NULL;
   mdl_object[cur_object].triangle = NULL;
   mdl_object[cur_object].uv_coord = NULL;
}
```

The code read the file and trying to compare the String in tmp with some specific headers in order to extract the information of each header and store them in the array mdl_object. The code continues for another two pages until everything in the ASCII file is collected in the array, and then the array can be utilized to draw the model which requires another loop for processing the array and starting creating the actual model on OpenGL side. These activities along the process are understandable by sophisticated 3D programmers, but time-consuming and nearly impossible for the programmers who are new to OpenGL to grasp and apply in short time. This is why the simpler approaches for creating scenery are introduced in the API

3.5 User Model

User model is the main dynamic models in the scene that can be followed by the camera, if the programmer prefers. The concept of adding the user model to the 3D world is similar to scenery, only the user model will always be the dynamic existing SketchUp model. The model is added by calling User constructor which requires file name, 2D position, offset in Z axis, scale, and rotation. Another different between the user model and the scenery is that the dynamic flag in the constructor is no longer needed, since the model is dynamic.

```
User drone = new User("drone.skp",//model file name
             new Point2f(30, 80), //position
             0, //offset Z
             0.2, //scale
             90); //rotation
terrain.addModel(drone);
```

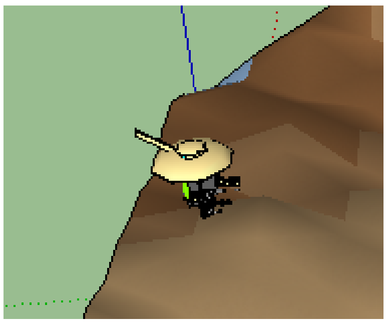


Figure 28. User Model added to the Scene

Figure 28 displays a drone result on the terrain, after the code for adding the user has been executed. The complexity of the similar process of importing the existing model in other OpenGL-based APIs was already described in the previous section. The programmers need to export the model from the 3D modeling software in ASCII format so that it can be read by OpenGL. Then, the information in the file is extracted, stored in the array, and the model can be drawn accordingly. The API simplifies all the complication and allows the novice programmer to add the model in a simpler, yet practical, way.

3.6 Animation

Animation is the process of generating sequence of movements and assigning to the dynamic models. The sequence of linear movements from one coordinate to the destination is represented by steps array containing the information at each step along the path, i.e. 3D coordinate, rotation, and animation time. There are three main steps of the animation, preparing steps arrays for each model, assigning the steps arrays to the model, and then initiating the animation. The following Java code demonstrates the utilization of the API for creating the animation for a dynamic car model as shown in Figure 29.

```
Step[] steps1 = terrain.genStepsArray(

        new Point2f(180, 100),//start
        new Point2f(10, 100),//destination
        0, 0,//start and stop angle
        1, 10);//start and stop time

car.setAnimation(steps1);
terrain.startAnimation();
```

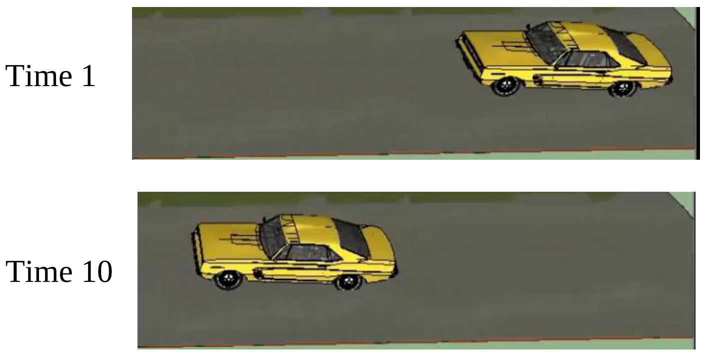


Figure 29. Animation Example

The result in Figure 29 indicates that the car translates in negative x axis with no rotation during the animation between time one and ten, as described in the steps array generation. The existing 3D APIs also support animation which requires the programmer to manually update the status of each model in the scene frame by frame. The following example presents simple C++ code using OpenGL for drawing an electron in an orbit and rotating it around the nucleus.

```
//Electron Orbit
glPushMatrix();
glRotatef(360.0f, -45.0f, 0.0f, 0.0f, 1.0f);
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
glTranslatef(x, y, z);
auxSolidSphere(6.0f);
glPopMatrix();

//Increment the angle of revolution
fElect1 += 10.0f;
x += 5.0f
y += 5.0f
y += 5.0f
if(fElect1 > 360.0f)
  fElect1 = 0.0f;
if(x > 100.0f)
  x = 0.0f
if(y > 100.0f)
  y = 0.0f
if(z > 100.0f)
  z = 0.0f
// Flush drawing commands
glFlush();
```

The electron's rotation is repeatedly incremented by ten degrees during the run time in order to make the electron circulates on its orbit around the nucleus. The electron orbit is also translated, along with the nucleus, in three dimension to reach the destination at (100, 100, 100), before the position is reset. Not much complexity is found in the code, but this is the animation of only one object in the scene. The purpose of this example is to show the concept of OpenGL animation which requires a manual update of the object's status. It will get much more complicated if there are more objects in the scene and longer sequence of animation is needed for each, for example, five or more cars in the scene. It would be easier to manage if the animation of each model can be prepared in advance and assigned to the model separately.

3.7 Animation Listener

Animation listener is a thread on Java side for watching the animation-related events on SketchUp side, e.g. collision, start, and end of animations, in order to detect and handle them. The key is to have SketchUp creates each event file in the file system and write the event data to the file. Then, the file will be detected by the thread which is implemented utilizing Java's WatchService [7] for monitoring the file system for new file creation. The new event object is created out of each new event file and the event handling method, defined in the Java program, can be called to deal with the event.

There are several types of events that might occur during the animation and collision between models is the one that is emphasized in this API. The collision happens once dynamic models run into each other during the animation runtime as demonstrated in Figure 30.

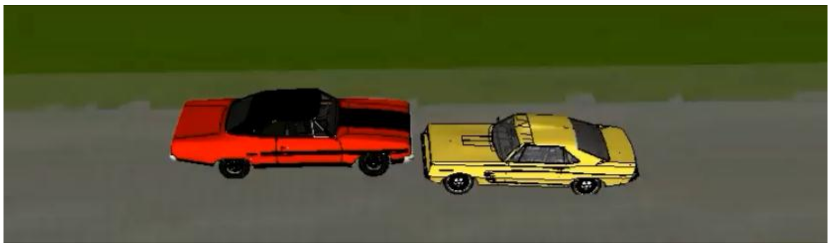


Figure 30. Collision between Dynamic Models

The collision detection in the API is automatically performed after the animation starts without requiring the programmers to implement it themselves. The reason for preventing the programmers from implementing the collision detection is that it is time-consuming and might be complicated for the novice programmers, which are aimed to be the main user group of this API. The following C++ code [28] shows the example of manual collision detection implementation using OpenGL.

```
void potentialCollisions(vector<BallPair> &potentialCollisions,
                         vector<Ball*> &balls, Octree* octree) {
  for(unsigned int i = 0; i < balls.size(); i++) {
     for(unsigned int j = i + 1; j < balls.size(); j++) {
       BallPair bp;
       bp.ball1 = balls[i];
       bp.ball2 = balls[j];
       potentialCollisions.push_back(bp);
     }
  }
}

bool testBallBallCollision(Ball* b1, Ball* b2) {
  //Check whether the balls are close enough
  float r = b1->r + b2->r;
  if ((b1->pos - b2->pos).magnitudeSquared() < r * r) {
     //Check whether the balls are moving toward each other
     Vec3f netVelocity = b1->v - b2->v;
     Vec3f displacement = b1->pos - b2->pos;
     return netVelocity.dot(displacement) < 0;
  }
  else
     return false;
}
```

In order to predict a collision between 3D balls, each potential collision pair of the balls need to be created in the method potentialCollisions(). Then, the collision between each pair is tested in the method testBallBallCollision() which checks if the balls are in range of the sum of their radius. Another test is performed in order to detect whether the balls' velocity vectors are moving towards each other. It can be noticed that the process of implementing collision detection in OpenGL is complicated and involves some mathematical theory which might be confused for the programmer who are new to understand. It would be better idea to provide automatic collision detection in the API and prevent the programmer from having to deal with the implementation themselves.

4. SKETCHUP INITIALIZATION

Since SketchUp only provides only Ruby API as its only interface, the problem is that SketchUp API needs to be generalized so that it can be used by other languages and the solution is Autoit. SketchUp is designed for human use i.e. multiple steps for sculpting 3D models using GUIs and Autoit is needed to change these steps into API commands. This section describes a process of initializing SketchUp employing Autoit.

Autoit can generalize SketchUp's Ruby API and make it available for other languages to use, especially Java. AutoIt is called through a Java interface to send commands to the SketchUp's Ruby console through four stages describing in Figure 31.

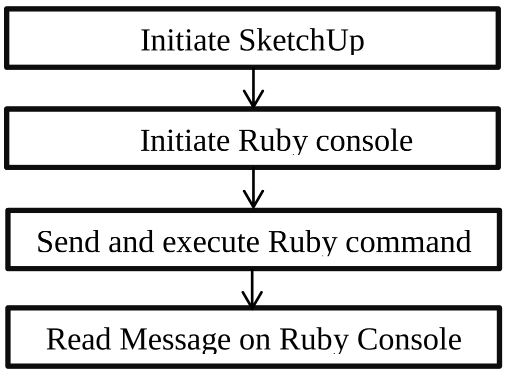


Figure 31. The Process of Automatically Executing Ruby command on SketchUp

Figure 31 shows sequence of tasks performing in order to send and execute the Ruby command. There are four steps for initializing SketchUp and opening its Ruby console; initiating SketchUp window, pressing start button, initiating Ruby console, and sending Ruby commands to the console. The first three steps can be performed by the API method startSketchUp()

```
SKPWrapper skp = new SKPWrapper();
skp.startSketchUp();
```

4.1 Initiate SketchUp & Ruby Console

SketchUp is opened in the method startSketchUp() which initiates the application accordingly to its path and then automatically clicks the button in order to start a new project. First, the path of SketchUp and names of each button of SketchUp's welcome interface, presented in Figure 32, are defined. Then, the method createWin() is called for starting the application and waits for the window with a specific title to appear. Finally, pressButton() is called in order to press a start button.

```
private static String SKP_EXE = "C:/Program Files
   (x86)/SketchUp/SketchUp 2013/SketchUp.exe";
private static String SKP_START_TITLE = "Welcome to SketchUp";
private static String START_BUTTON = "Start using SketchUp";
private static String SKP_TITLE = "Untitled - SketchUp";
createWin(SKP_START_TITLE, SKP_EXE);//initiate SketchUp
pressButton(SKP_START_TITLE, START_BUTTON);//click start button
```

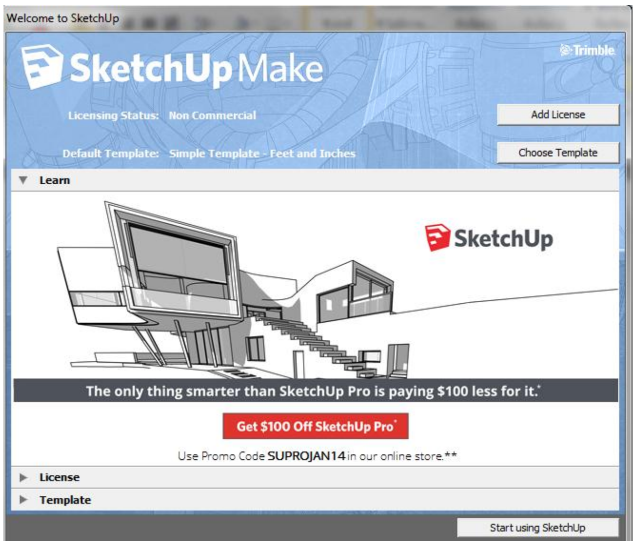

Welcome to SketchUp
SketchUp Make
Trimble
Licensing Status: Non Commercial
Add License
Default Template: Simple Template - Feet and Inches
Choose Template
Learn
SketchUp
The only thing smarter than SketchUp Pro is paying $100 less for it.*
Get $100 Off SketchUp Pro*
Use Promo Code SUPROJAN14 in our online store.**
License
Template
Start using SketchUp

Figure 32. SketchUp Startup Window

After SketchUp is started, the button “Start using SketchUp” is pressed in order to create a new project. There are two parameters required for this action: the first one is button name and another is the name of window in which the button consists. A new project in SketchUp is initiated after the button is pressed as displayed in Figure 33.

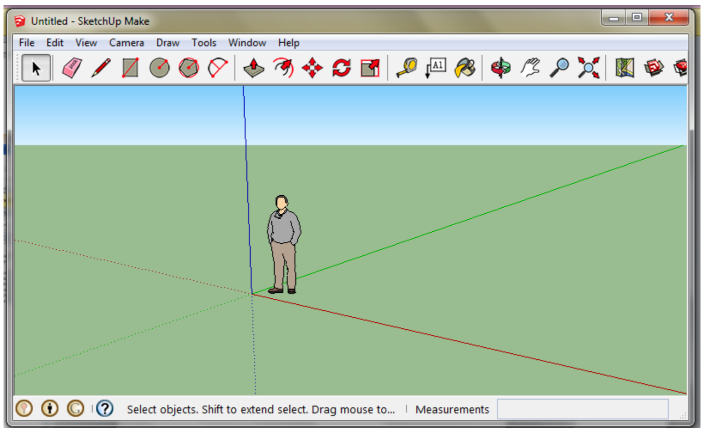

Untitled - SketchUp Make
File Edit View Camera Draw Tools Window Help
Select objects. Shift to extend select. Drag mouse to...
Measurements

Figure 33. A New Project of SketchUp

Figure 33 shows a general user interface of SketchUp appearing once a new project is created. The interface contains multiple tools and menus for the user to select and utilize. However, our goal is to automatically generate an animated 3D scene by sending and executing Ruby commands in SketchUp's Ruby console.

4.2 Initiate Ruby Console

Ruby console can be opened in SketchUp by pressing hotkey alt+w followed by alt+r. This can be applied with Autoit in order to start up the console. The following code is in method sends key press inputs to SketchUp for opening the Ruby console.

```
private static JAutoIt lib = JAutoIt.INSTANCE;//create instance
 lib.AU3_Send("{ALT down}wr{ALT up}", 0);//hold ALT press 'w'and'r'
```

The last parameter of AU3_Send() indicates the type of key process: 0 represents default mode which the text will contain special while - 1 means keys will be sent raw. The following picture shows the result after Ruby console is started.

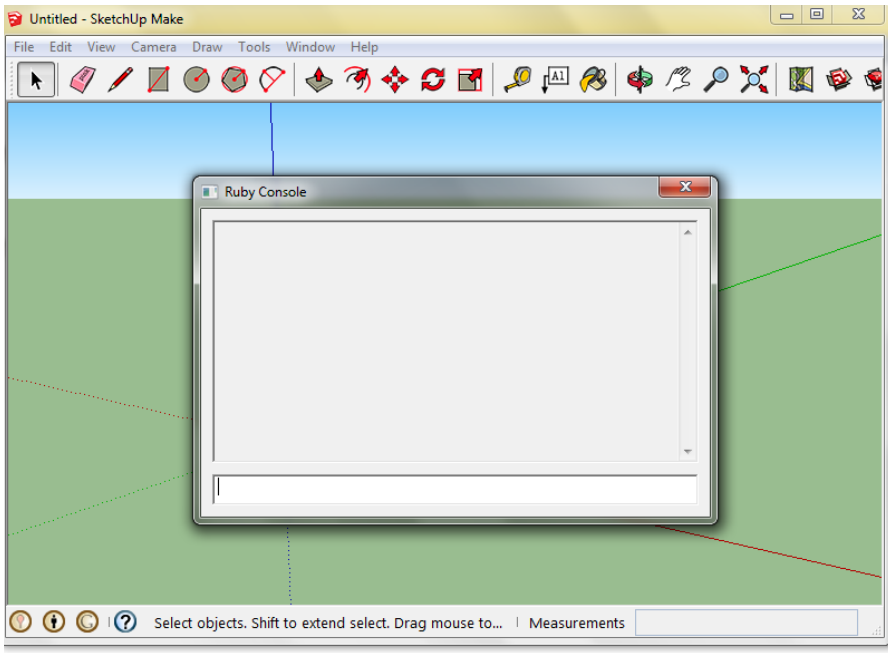

Untitled - SketchUp Make
File Edit View Camera Draw Tools Window Help
Ruby Console
Select objects. Shift to extend select. Drag mouse to...
Measurements

Figure 34. Ruby Console Initiated

Figure 34 shows that now the ruby console is opened by sending key press alt+w followed by alt+r to SketchUp. The next step is sending Ruby command to be executed on Ruby console which is explained in next section.

4.3 Send and Execute Ruby Command

Ruby console in SketchUp consists of two parts: Edit1 and Edit2. According to Figure 35, Edit1 is input text field and Edit2 is the feedback display. Ruby commands are and executed at Edit1. Then the result will display at Edit2.

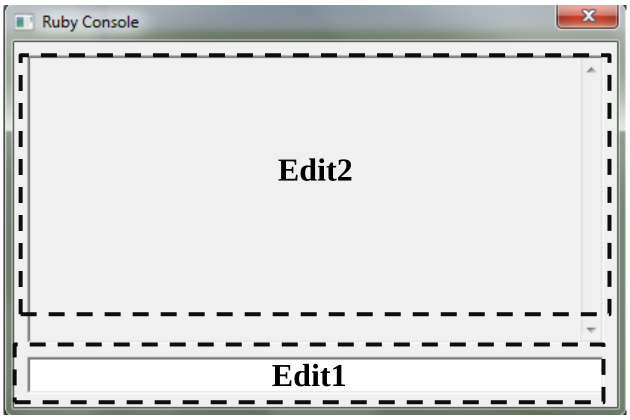
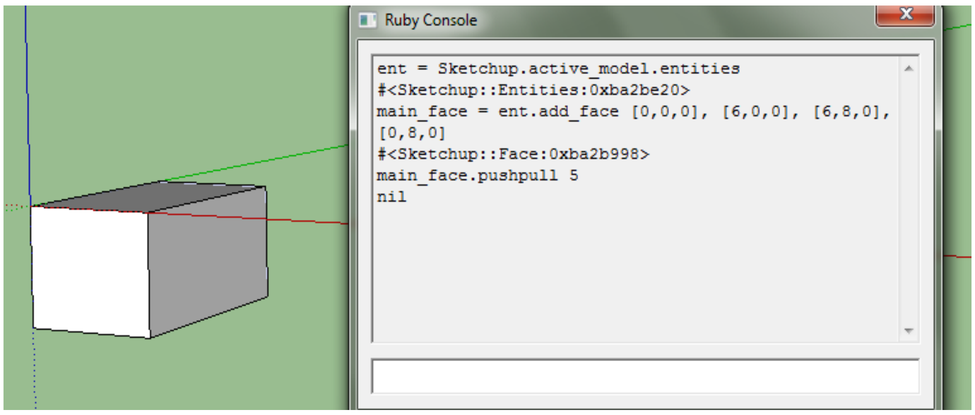


Figure 35. Ruby Console Consisting of Two Parts

Before sending anything to Edit 1, the part has to be selected first. Then the method AU3_Send() is used to send ruby command in a form of String to Edit1 in Ruby console. In this case, a code for box generation from section 4.3 is sent (by lib.AU3_Send()) followed by pressing 'Enter' for executing the command. Then the result displays on Edit2 as shown in Figure 36.

```
private static final String RC_TITLE = "Ruby Console";
lib.AU3_ControlFocus(RC_TITLE, "", "Edit1")
//send box generation commands
```



```
ent = Sketchup.active_model.entities
#<Sketchup::Entities:0xba2be20>
main_face = ent.add_face [0,0,0], [6,0,0], [6,8,0],
[0,8,0]
#<Sketchup::Face:0xba2b998>
main_face.pushpull 5
nil
```

Figure 36. The Result on Edit2 after Executing Ruby Commands

Figure 36 shows the 3D box result generated from Ruby commands that are sent and executed in the ruby console. The messages from each execution indicating, whether the execution is successful or not, are displayed in Edit2 are read back to Java program. The approach for retrieving the messages to Java is explained in the next section.

4.4 Read Message on Ruby Console

The process of returning the feedback messages back to Java is implemented in method getDisplay(). Autoit API in Java provides method AU3_ControlGetText() which allows user to get the text on the specific window. The method is employed for retrieving the feedback from Edit2. At the beginning, the array of character is defined. Then AU3_ControlGetText() is called in order to read text on Edit2 section in ruby console and stored in the array. Then, the array is converted to String and printed out.

```
char[] display = new char[200];
lib.AU3_ControlGetText(RC_TITLE, "", "Edit2", display,
      display.length);
String displayText = AUUtils.chars2String(display);
System.out.println("----------- Display -----------");
System.out.println(displayText);
System.out.println("-------------------------------");
```

SketchUp, along with its Ruby console, is now initiated and Ruby commands for 3D objects creation can be sent to the console. SketchUp's Ruby API is now generalized utilizing Java interface for Autoit. The following chapter explains algorithms and practical uses of the API to generate each component of the animated 3D virtual world.

5. LANDSCAPE GENERATION

Landscape represents a main static component of the scene. It is the model on which other dynamic and static models will be positioned. The API allows two types of landscape to be generated; fractal landscape generation and landscape generation from file. The process of generation and texturing of terrain focuses on not only simplicity for beginners, but also capability of achieving interesting results. The API hides algorithm for producing a terrain but allows the programmer to adjust the parameters of the landscape in Terrain class.

This chapter describes alternative approaches of landscape creation including fractal terrain generation from Diamond-Square algorithm [25], terrain generation from heightmap and coordinates file. These features are designed and added to the API in order to provide the programmer options and a control over the terrain appearance.

5.1 Fractal Terrain Generation

Fractal terrain generation generates random terrain results by repeatedly performing Diamond-Square algorithm [25] in order to repeatedly create small fragments of the terrain until the terrain is completed. Both flat and rough terrains are supported by fractal terrain generation. By altering the constructor parameters, different random terrains can be generated. For example, a flat terrain can be created by assigning zero to terrain's depth, while a rough terrain is produced by assigning positive depth to the constructor. The results of both textured flat and rough terrains are demonstrated in Figure 37.

```
public Terrain(
  int depth, //terrain’s depth
  int resolution, //resolution = 2^size+1
  int plgSize, //size of each polygon
)
```

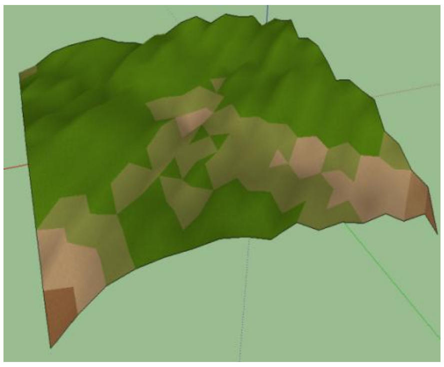


Figure 37. Fractal Terrains

There are four steps for producing the fractal terrain; generate coordinates of the terrain, load coordinates to SketchUp, create polygons mesh out of the coordinates, and texture the terrain. Figure 38 shows a diagram demonstrating the process of generating fractal terrain.

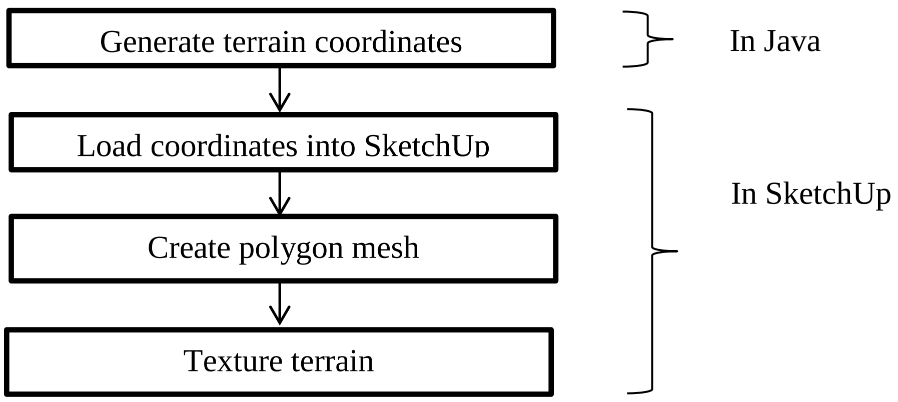


Figure 38. Four Steps for Fractal Terrain Generation

The process of generating terrain occurs on both Java side and SketchUp side. The preparation of terrain coordinates involves complex calculation which is fast to perform on the Java side, and then the process of generating visual result of the terrain and texture the terrain is performed on SketchUp.

5.1.1 Generate Terrain Coordinates

Terrain coordinates are generated utilizing Diamond-Square algorithm [25]. The concept of the algorithm is to iteratively assign random height value to each node of a 2D array representing the terrain. All nodes in the same area are assigned with similar random value so that the terrain is continuous and smooth

Diamond Step

The diamond step consists of taking the four points arranged in a square shape, averaging the four values and adding a random value to the average in order to create a diamond shape. The method sampleDiamond() describes a behavior of diamond step.

```
public void sampleDiamond(int x, int y, int size, double value){
  int hs = size / 2;
  double a = sample(x - hs, y);
  double b = sample(x + hs, y);
  double c = sample(x, y - hs);
```

```
    double d = sample(x, y + hs);
    setSample(x, y, ((a + b + c + d) / 4.0) + value);

}
```

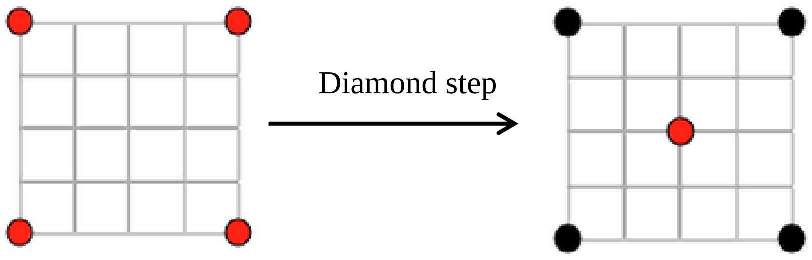


Figure 39. Terrain after Performing Diamond Step

According to Figure 39, diamond step creates a new coordinate out of four neighbor coordinates at its corners. Height of each coordinate at the corner is extracted by the method sample(). Then a height of the new coordinate can be assigned by an average height of these neighbor coordinates added with a random small value.

Square step

The square step sets four midpoints between corners by averaging two values on each side of the square, then slightly smaller or bigger random value than existing neighbor points is assigned to each midpoint created. Figure 40 shows the terrain result after Square step is applied.

```
public void sampleSquare(int x, int y, int size, double value){
    int hs = size / 2;
    double a = sample(x - hs, y - hs);
    double b = sample(x + hs, y - hs);
    double c = sample(x - hs, y + hs);
    double d = sample(x + hs, y + hs);
    setSample(x, y, ((a + b + c + d) / 4.0) + value);
}
```

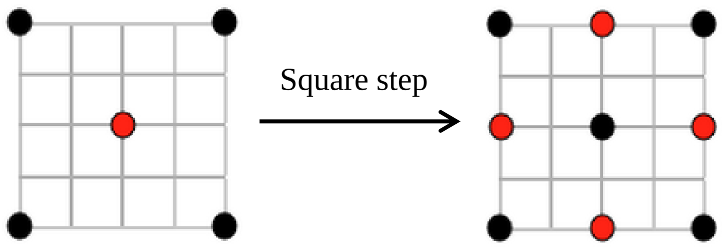


Figure 40. Terrain after Performing Square Step

After the square step in Figure 40, a big single square represented the terrain is now divided into four sub squares and the height values of coordinates at the corners of each square are set. Then the diamond and square step are repeated until each coordinate in the grid has height value assigned as shown in Figure 41.

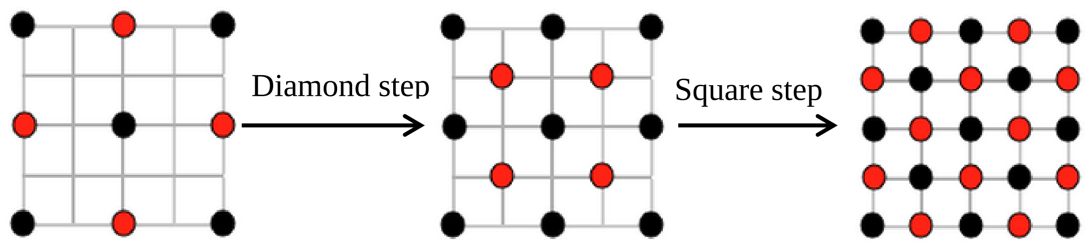


Figure 41. Repetition of Diamond and Square Steps

As the repetition process goes, the points in the grid are being assigned the value until there's no empty point left on the grid. After the coordinates in the grid are ready, they will be written to the text file.

```
BufferedWriter out = new BufferedWriter(new FileWriter(fileName));
//write resolution to the first line then then write coordinates
for (int i = 0; i < square.length; i++) {
   for (int j = 0; j < square.length; j++) {
      out.write(i + "," + j + "," + (int) square[i][j]);
      out.newLine();
   }
}
```

After a resolution of the terrain is written to the first line, the coordinates stored in 2D array 'square' are written to the text file in the nested loop. The index i and j represents the x and y value of each coordinate, while z is extracted from the coordinate array. Each coordinate is written as a single line of the file and once the process finishes, the file will be read by Sketchup in order to start creating terrain model.

5.1.2 Load Coordinates into SketchUp and Create Polygon Mesh

Once the coordinates are prepared on the Java side and written to the text file, the process of reading the file and create a polygon mesh is performed on SketchUp side. This is the part where the set of terrain coordinates are converted into a visual from of a plain terrain model as displayed in Figure 42.

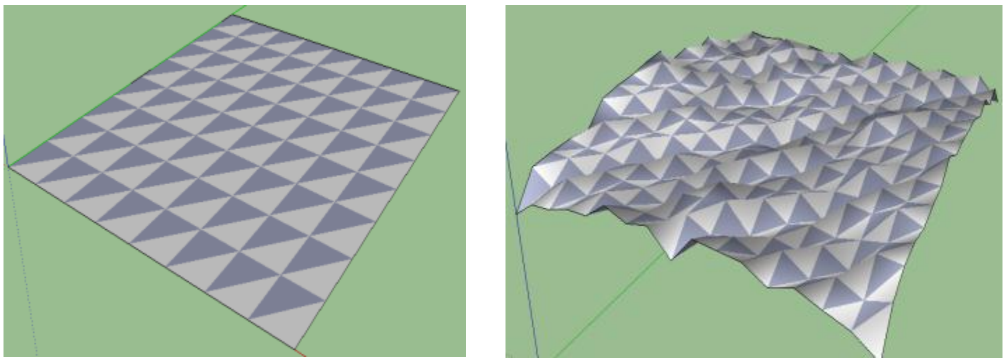


Figure 42. Flat and Rough Terrains

The file is read into SketchUp line by line using Ruby script. In a loop, the resolution of terrain is extracted out of the first line, then x, y, and z values in other lines are split by comma and stored in 2D array ‘square’. The x and y represents indices of element and z is the value of the element. The following Ruby script extracts and stores the values in each line in 2D array.

```
for line in lines
   if first line
      #extract terrain resolution
   elss
      buff = line.split(",")
        x = buff[0].to_i
        y = buff[1].to_i
        z = buff[2].to_i
        square[x][y] = z
   end
end
```

After 2D array is completed, the process of creating polygon mesh is started. The polygon mesh, which can be used to produce face or surface such as terrain, consists of many polygons, so the concept is to form each polygon using three neighbor coordinates and then add them to the mesh over and over until the mesh is completed. Figure 43 shows that two polygons are created out of four coordinates. The following Ruby code demonstrates the behavior of a polygon generating loop in each iteration.

```
iup = i+1
jright = j+1
#const is the polygon’s width
pt0 = Geom::Point3d.new i*const, j*const, square[i][j]
pt1 = Geom::Point3d.new i*const+const, j*const, square[iup][j]
pt2 = Geom::Point3d.new i*const, j*const+const, square[i][jright]
pt3 = Geom::Point3d.new i*const+const, j*const+const,
      square[iup][jright]
mesh.add_polygon pt0, pt1, pt2 #add a polygon to polygon mesh
mesh.add_polygon pt3, pt1, pt2
```

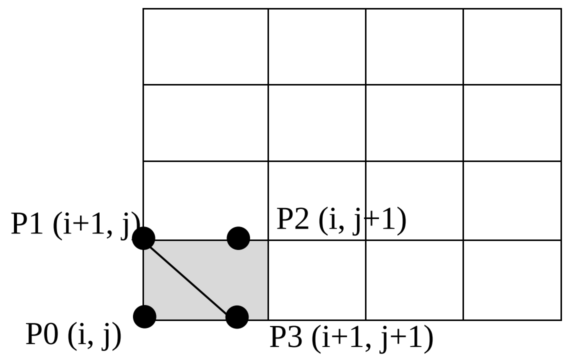


Figure 43. Two polygons Creation in Each Iteration

Through nested loop, for each iteration, two polygons are created and added to the mesh. According to Figure 43, the first polygon consists of three coordinates; (i, j), (i, j+1) and (i+1, j) and the second polygon is from coordinate (i+1, j), (i, j+1) and (i+1, j+1). In practice, i and j values represent the index of 2D array which contains the height value of coordinate (i, j). The polygon is formed by different 3D points which mean they could have different height and orientation. The process continues until there is no coordinates which are not member of the polygon left.

5.1.3 Texture the Terrain

Textures are necessary for giving realism to the objects in the scene. On the real natural landscapes, there are several different texture levels and materials, for example, a mountain hill surface consists of water and sand while the grassy hill contains grass and sand textures. This chapter explains the approach for applying multiple textures to the terrain and creating a transition between each texture level of the terrain.

The concept of terrain texturing is to separate the polygons into different texture levels, accordingly to their heights. Then the texture can be applied to each level differently. The API makes it available for the programmer to define the number of texture levels and the texture of each. Then it will generate transition between each texture level in order to make the textures more continuous. Each transition is a merged texture of the textures from upper and lower levels. The method for terrain texturing, similar to the terrain generation, still aims on ease of use for novices.

The first two steps of fractal terrain generation are automatically performed after new Terrain instance is initiated by the programmer. The following Java code example demonstrates how to texture the terrain, after a plain terrain is generated, by creating three texture levels and mapping water, dark sand, and sand texture in JPEG format to each level consecutively. The result of the code is displayed in Figures 44.

```
terrain.setLevel(3);//set number of texture levels
terrain.setTexture(1, "water.jpg", 50);// material at each level
terrain.setTexture(2, "darkSand.jpg", 25);
terrain.setTexture(3, "sand.jpg";, 25);
terrain.texture();//start texturing
```

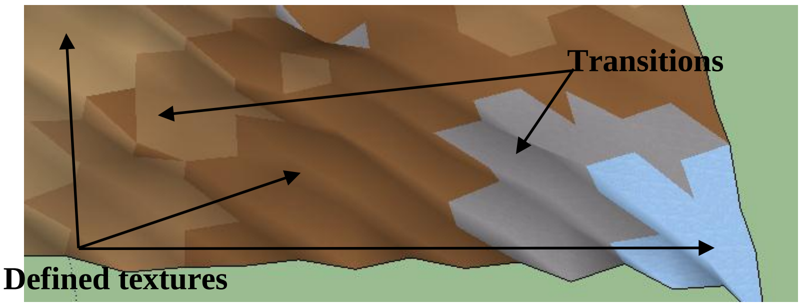


Figure 44. Textured Terrain

The textured terrain result in Figures 44 indicates that, from three defined texture levels, there are two transition levels generated and inserted between each level by the API. It can be concluded that for every N texture levels, there will be N-1 transition levels generated. The terrain texture looks continuous and natural. Figure 45 shows a diagram representing terrain multi-texturing paradigm.

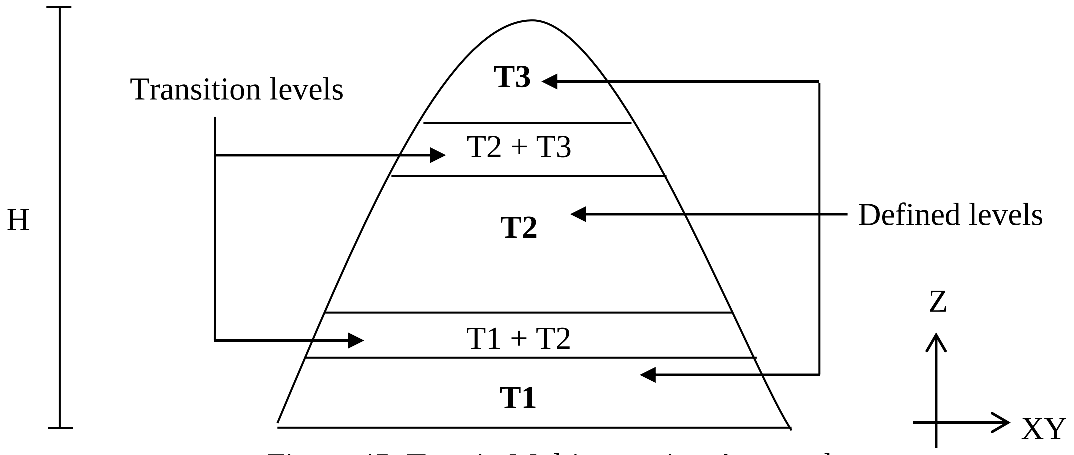


Figure 45. Terrain Multi-texturing Approach

According to Figure 45, level T1, T2 and T3 represent each texture level of a terrain. Their material and heights are defined by the programmer. Other hidden levels; T1+T2 and T2+T3 are automatically generated by the API in order to create transitions between existing texture levels. The materials of these hidden levels are automatically generated by blending the material from the upper level with the material from the lower level. After textures are prepared, they can be applied to the terrain.

There are three methods for terrain texturing on the Ruby side; setNumLevel(), setMaterial(), and applyTexture(). The first method receives the number of texture levels and creates an array for textures accordingly to the given number.

```
def setLevel(num)
   $level = num
   $tex = Array.new(num)
end
```

The height array are initiated when the user defines the levels of terrain's textures by calling setLevel(). On the Java side, there is an array for storing the heights as well. The height array on Java are used to calculate the hidden levels' heights, after the heights of the main levels are defined by the programmer.Figure 46 shows an example of the value in an array after setting the heights of three texture levels.

```
int[] height;
public boolean setTexture(float num, String texName, int height){
   int index = (int) (num*2 - 2);//calculate the true index
   this.height[index] = height;
   lib.AU3_Send("terrain.setMaterial(" + index +", \""+ texName +
   "\","+ height +"){ENTER}", 0);
   return true;

}
```

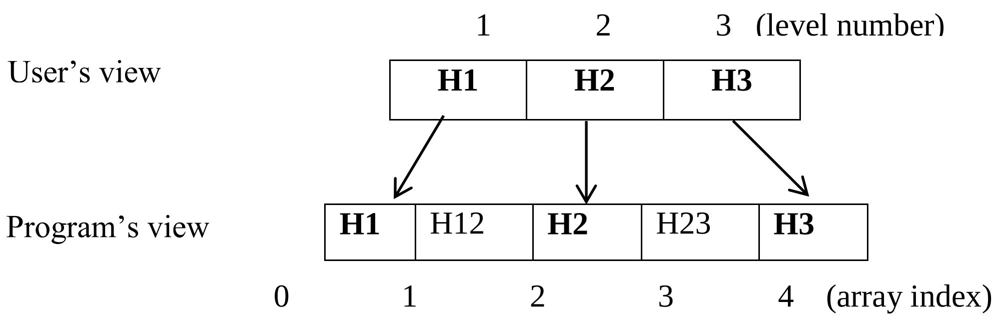


Figure 46. The Height Arrays from User's View and API's View

After the programmer finishes with assigning the heights of to each texture level, the next step is to have the API calculates the height for each hidden level and store them to the array from Figure 46, once the Java method texture() is called. The height of each hidden level (H12 and H23) will be calculated accordingly to the neighboring levels' height.

Hxy = (Hx/2 + Hy/2)/2
H12 = (H1/2 + H2/2)/2
H23 = (H2/2 + H3/2)/2

The code for calculating and setting the hidden levels' height is inserted to the texture method, on the Java side.

```
  public boolean texture() {

      int height;
      for(int i = 1 ;i<tex.length-1; i+=2){
          //create merged texture
          tex[i] = blendTex(tex[i-1], tex[i+1]);
          //calculate the hidden level's height
          height = (this.height[i-1]/2 + this.height[i+1]/2)/2;
          setTexture((float)(i+2)/2, tex[i],height);

      }
      //send Ruby command to start texturing terrain on SketchUp side
      lib.AU3_Send("terrain.applyTexture(){ENTER}", 0);
      return true;

   }
```

The method processes the height array and fills an empty slot (hidden level H12 and H23) by an average value calculated of the height at i+1 and the height at i-1. After that the method sends a method calls for setting the textures of hidden levels and texturing the terrain. Figure 47 shows an example of the array after the height of each level is calculated

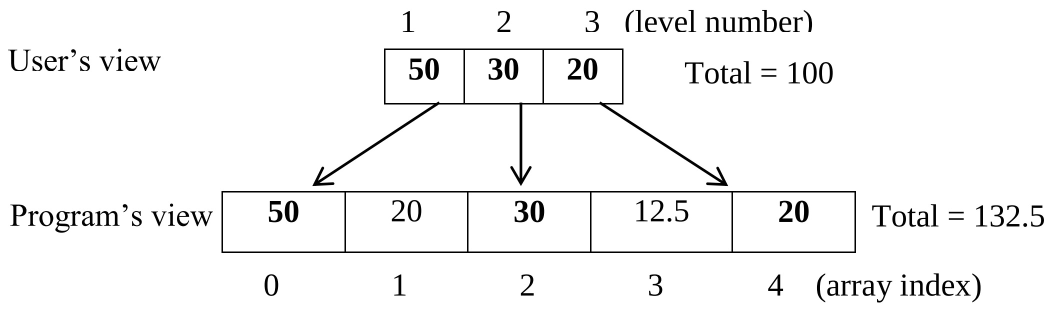


Figure 47. The Completed Height Arrays

According to Figure 47, the total height of terrain texture levels in the user's view is 100, but the real total height in the program's view is 132.5. However, the value in the array cannot be directly used to apply the textures, since the value in the array can only specify the proportion of each texture level in the terrain. The real height for each texture is needed to be calculated next, relatively to the distance between the highest face and the lowest face of the terrain.

Once the method applyTexture() is called on the Ruby side, it begins to calculate the real height of each level accordingly the distance between the lowest and highest face. In order to do so, the total height and the height array on the Ruby side is used.

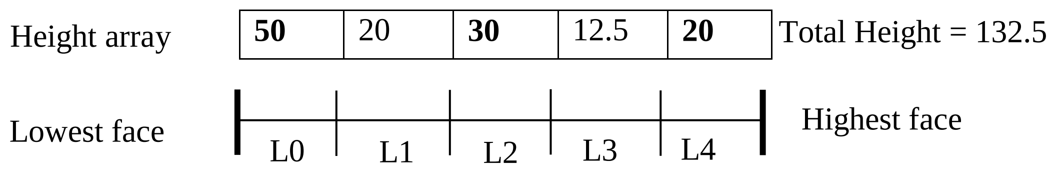


Figure 48. Terrain Levels Divided from the Highest and Lowest Faces

According to Figure 48, assume that the lowest face is at Z = 80 and the highest face is at Z = 150, the distance between these face is 70. The real height of each level; L0, L1, L2, L3, and L4 can be calculated proportionally to the value given in the height array using the following formula.

Ln = (face_high – face_low)*(Hn/totalHeight)
L0 = 70 * (50/132.5) = 26.4
L1 = 70 * (20/132.5) = 10.6
L2 = 70 * (30/132.5) = 15.9
L3 = 70 * (12.5/132.5) = 6.6
L4 = 70 * (20/132.5) = 10.5

The real height value of each level is set to the height array on the SketchUp side, and then the array is ready to be used to apply textures to the terrain. Here is the Ruby code for processing each face and applying texture accordingly to the height array in the applyTexture() method.

```
upper = face_high #upper bound of the highest level
$faces.each do |a|#process each polygon in the terrain

    cp = a.bounds.center[2] #get the Z value
    if cp == face_high
        a.material = $tex[$level-1]
        a.back_material = $tex[$level-1]
    elsif cp == face_low
        a.material = $tex[0]
        a.back_material = $tex[0]
    else

      ($dist.length-1).downto(0) {|i|
        if cp < upper
         if cp >= (upper - $dist[i])#lower bound of highest level
           a.material = $tex[i]
           a.back_material = $tex[i]
         end
        end
        upper -= $dist[i]#upper bound of the lower level
      }
      upper = face_high #reset the upper bound for the next face
    end
 end
```

The loop processes and extracts the height out of each face in the terrain. The height is employed to check for which texture level the polygon falls in. After a specific texture level is found, the right material will be selected and applied to the face. The loop continues until each polygon in the terrain is correctly textured.

The process of fractal terrain generation is now finished. The terrain coordinates are first generated from Diamond-Square algorithm which supports both flat and rough terrain. Then they are written to the text file, loaded to SketchUp, and utilized to create a polygon mesh. Then the terrain can be textured. The next section delineates another alternative approach of generating the terrain from existing heightmap.

5.2 Terrain Generation from Heightmap

The API provides another feature for extracting RGB value out of each pixel of grayscale heightmap and creating a terrain model from these values. This not only allows the programmers to be able to easily design the terrain model by drawing and supplying a heightmap to the API, but also makes it possible to create a combination result of both flat and rough terrains shown in Figure 49.

The only difference between terrain generation from fractal algorithm and heightmap is in the process of terrain coordinates creation, while the texturing methods are still the same as in the previous section. This section explains how to create the terrain from heightmap and and leaves out the texturing methods.

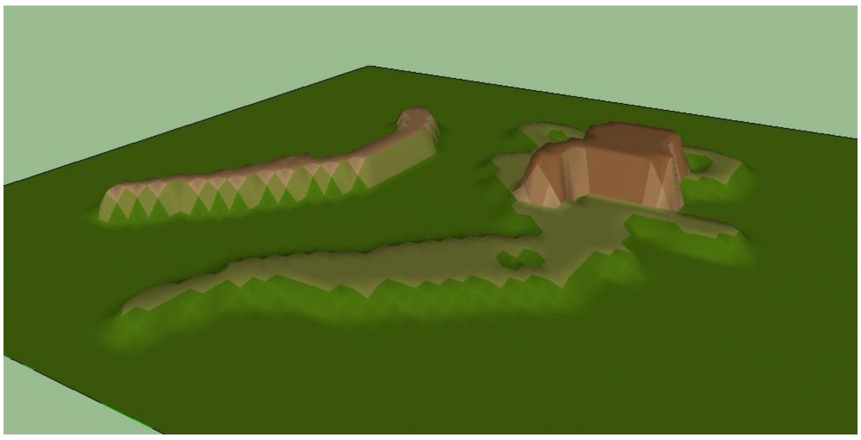


Figure 49. Terrain Model Generated from Heightmap

Figure 49 represents a result of textured terrain model generated from the heightmap approach on SketchUp side. The result shows that there are both rough and flat regions on the model. The API allows the programmers to create and adjust parameters of the terrain, including terrain's depth and polygon size provided in the following Terrain constructor.

```
public Terrain(String file, int depth, int plgSize){
     try {
        int i = 0, j = 0;
        BufferedImage image = ImageIO.read(new File(file));
        square = new double[image.getWidth()][image.getHeight()];
        for (int y = 0; y < image.getHeight(); y++) {
           for (int x = 0; x < image.getWidth(); x++) {
               int height = depth - (int) ((image.getRGB(x, y) &
                            0xFF) / 255.0 * depth);
               i = x;
               j = image.getHeight()-1-y;
               square[i][j] = height; //square[x][y] = z
           }
        }
        //write coordinates (x, y, z) to text file
     }catch (IOException e) {
        System.out.println(e);
     }
     createPolygonArray()
     //load coordinates into SketchUp and create terrain model
}
```

The constructor requires terrain grayscale heightmap file, terrain's depth, and size of each polygon as arguments. Since the RGB values of each pixel in the heightmap are the same, the value representing that same RGB of each pixel will be extracted and stored in an array. Before the coordinates are written to coordinates text file, they are utilized to create an array of polygons (QuadArray instances) by calling createPolygonArray() which repeatedly generates two polygons out of each four neighbor coordinates and stores in the polygon array. The array is used later for calculating z position of the model on the terrain. At the end, the coordinate file is read by SketchUp and the terrain model can be relatively generated.

The first parameter of the constructor, heightmap file name, originates how the terrain model looks by defining height of each region on the terrain model as shown in Figure 50.

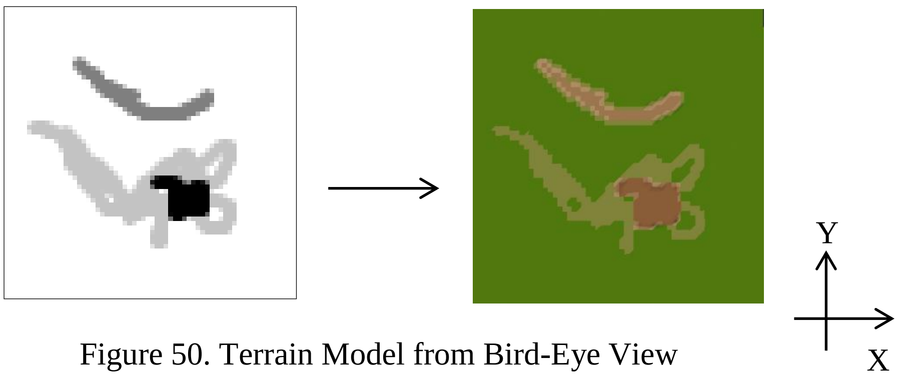


Figure 50. Terrain Model from Bird-Eye View

It is demonstrated in Figure 50 that the terrain result on the right picture is consistent with the heightmap on the left. Height of each polygon in the terrain is relative to color in the grayscale image. The polygon height, representing each pixel, starts at zero if the pixel is white and tends to increase if the pixel color gets darker and closer to black.

Aside from the image file, other parameters also have an influence on the result as well. The terrain depth defines a distance between the lowest polygon and the highest polygon in the terrain. The depth is utilized to calculate height of each polygon relatively to the following equation.

```
    height = depth - ((RGB / 255) * depth);
```

For the grayscale images, R, G, and B value will be the same. RGB value in the equation represents that same value. The RGB is divided by 255 which is the maximum value in order to calculate a scale of the pixel color in range between zero and one. For instance, if the color is white (RGB = 255), the calculated height will be zero. The result of terrain model with different depth is presented in Figure 51 and Figure 52.

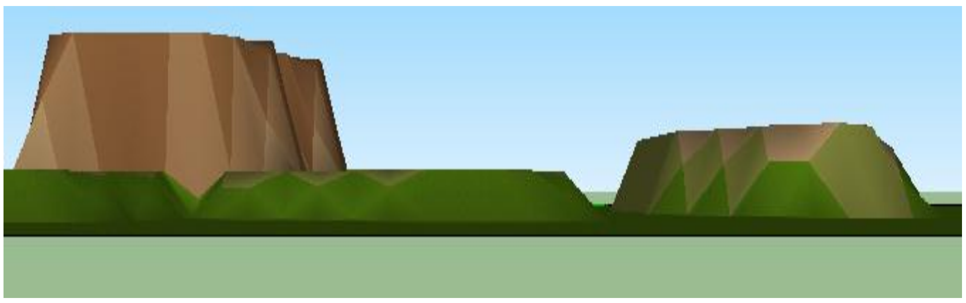


Figure 51. Terrain Model of Depth 100

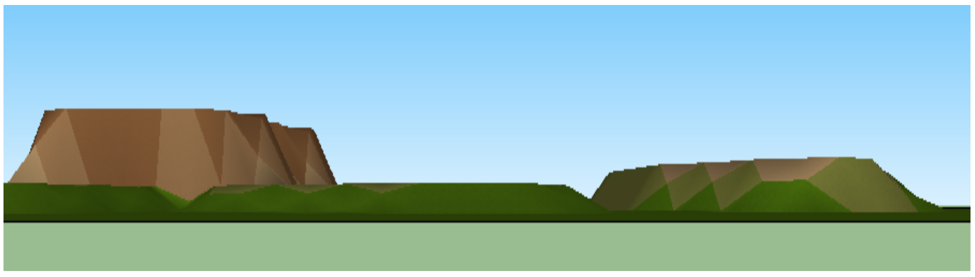


Figure 52. Terrain Model of Depth 50

Figure 51 and 52 show a different terrain results generated with different depths. It can be noticed that the height of terrain hills decrease as the depth is reducing. Another argument that controls the terrain appearance is polygon size which defines width of each polygon member in the terrain. The terrains of alternative polygon sizes are shown in Figure 53.

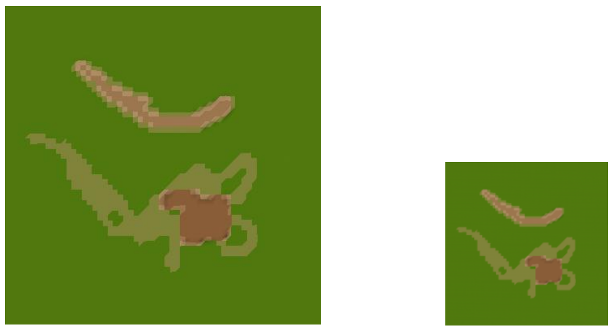

a) Polygon size= 30 b) Polygon size= 15

Figure 53. Terrain Models of Different Polygon Sizes

Polygon size affects the size of the terrain model directly, according to Figure 53. Similar to the terrain's depth, the terrain result gets bigger if the polygon size increases and vice versa. Aside from utilizing heightmap, there is also another way of generating the landscape using the existing coordinate file describing in next section.

5.3 Terrain Generation from Coordinates File

Assuming that the programmer prefers to randomly generate terrain using Diamond-Square algorithm instead of heightmap, it can be done by supplying terrain depth, terrain resolution, and polygon size to the Terrain constructor. This approach generates set of coordinates, representing different terrain each time the program is executed, before writing them to the text file.

Due to the fact that the terrain result from Diamond-Square algorithm is random and the terrain model in the next execution of the program might not be the same, the API provide another alternative Terrain constructor in order to allows the programmer to create the terrain from the existing coordinate file.

```
public Terrain(String file, int plgSize){

     String line;
     String[] info;
     int count = 0;
     int size, x, y, z;
         //create new BufferedReader
     while ((line = br.readLine()) != null) {

         if (count==0){//first line
             //extract terrain resolution and store in ‘size’
             square = new double[(int)Math.pow(2,size)+1]
                                 [(int)Math.pow(2,size)+1];
         }
         else{ //not first line
```

```
                //extract x, y, z from the line
                square[x][y] = z;
            }
            count++;
        }
        createPolygonArray();
        //send Ruby command for generating terrain from the file
    }
```

The constructor requires a coordinate file and a polygon size, before begins the process of reading the file and generating terrain model on SketchUp. If the first line, which stores terrain resolution, is read, a resolution of terrain will be extracted and employed to defined size of 2D array for storing coordinates. Then the coordinate information is extracted from other lines and stored in the array. At the end, the polygon array will be created and Ruby commands for generating terrain will be sent to SketchUp.

The programmer now has alternative ways of producing landscape from heightmap. The terrain results from both approaches are likely to be more controllable, comparing with terrain generation by Diamond-Square algorithm, described in previous reports, which provides more random terrain model. However, this depends on the programmer needs. The next chapter explains an update on collision recovery method provided by the API to recovery collisions between models.

6. SCENERY MODELS

Scenery is another interesting feature of the API. The scenery models represent features of the landscape and define a location of a particular scene. The API emphasizes the advantages of SketchUp 3D Warehouse by providing the programmer an option to import existing 3D models from the warehouse and other sources. Moreover, the programmer can also create and add to the scene some basic geometry shapes as displayed in Figure 54.

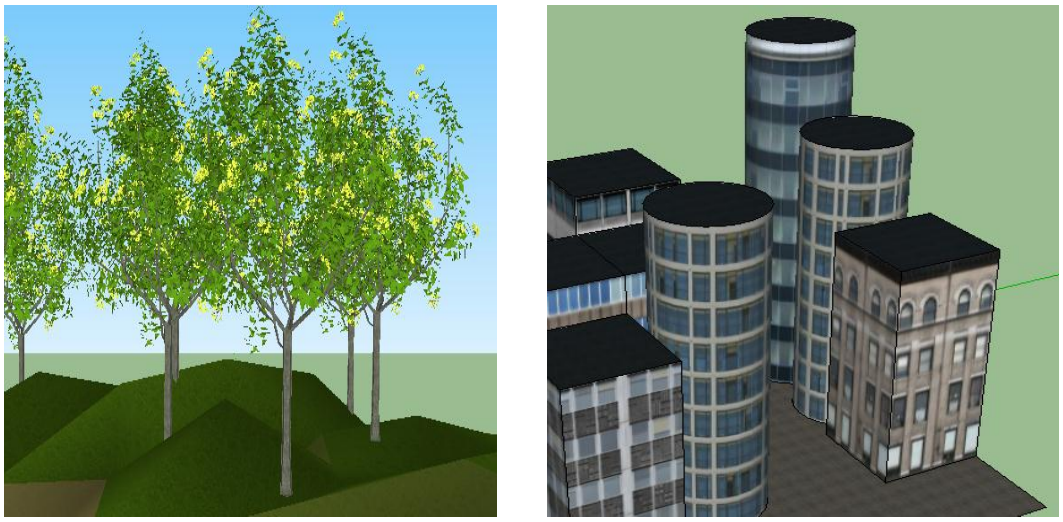


Figure 54. Scenery Models and Shapes

This chapter goes through the methods of adding scenery models to the scene including existing models importing, geometric shapes generation, and scenery generation from text file.

6.1 Existing SketchUp Models

The existing SketchUp models can be downloaded and imported to the scene from SketchUp 3D Warehouse which contains variety of shared models designed and sculptured by SketchUp users all over the world. This type of scenery models are divided into two categories; static and dynamic scenery models. Static scenery models symbolize the models that cannot move, e.g. trees and traffic lights, while dynamic scenery models instantiate movable scenery models in the scene, e.g. cars and floating robots. The following code demonstrates how the static models, in Figures 55, are imported to the scene.

```
Model tree = new Model(
                "tree1", //model name
                "tree.skp", //SketchUp model file name
                new Point2f(100,100), //position
                0, //offset in Z axis
                1, //scale
                0, //rotation
                false //not dynamic model
              );

terrain.addModel(tree);
```

The API requires the programmer to specify model name, model file name, 2D position, Z offset, rotation, and whether the model is static or dynamic by changing the flag in the Model constructor. Once the Model instance is constructed, it will be add to the scene by calling Terrain’s method addModel() in Appendix B.4.

```
public boolean addModel(Model model) {

  Point2f position = model.getPosition();
  float z = getHeight(position.x, position.y) +
            model.getOffsetZ();
  //extract data and send with Ruby commands to SketchUp
}
```

In addModel(), the API automatically calculates a position of the model on the terrain surface in Z axis using Terrain’s method getHeight() from Appendix B.4, before constructing it by sending Ruby commands to SketchUp. The method getHeight() finds a specific polygon, of the terrain, where the given 2D coordinate falls in and then returns a height at the middle of the polygon calculated using Barycentric approach [29]. This allows the model to be perfectly positioned to the terrain surface without the programmer trying to define its Z value. However, if the programmer wants to elevate the model above the terrain, it can be done by defining Z offset in the Model constructor. The result of the model defined in the example constructor is presented in Figure 55.

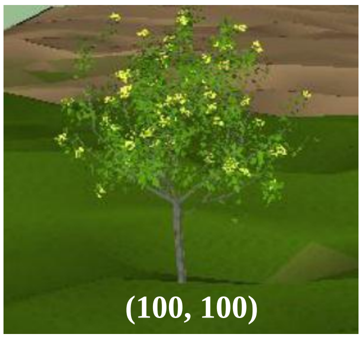


Figure 55. Static Model Added to the Scene

The tree model in Figures 55 indicates the result from the code example for adding a static model. In this case, the model is static and positioned at (100, 100) without no rotation and resizing. The next example is for adding dynamic car model to the scene by changing the last parameter, which indicates if the model is dynamic, to true. The result is contained in Figures 56.

```
Model car1 = new Model ("car","blueCar.skp",
               new Point2f(170,175),
               0,
               0.2,
               0,
               true //dynamic model
             );
terrain.addModel(car1);
```

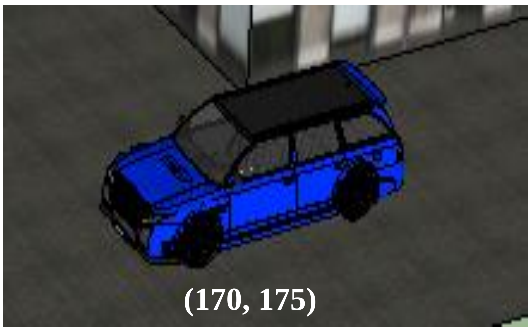


Figure 56. Dynamic Model Added to the Scene

The dynamic blue car is imported to the scene relatively to the defined preference. The difference between static and dynamic model is that the dynamic model can be assigned with animation in terms of steps arrays which are sequence of movements for the model. Then, the dynamic model will move accordingly, after the animation starts.

6.2 Geometric Shapes

In SketchUp, several kinds of shapes can be constructed and added the 3D scene. The API allows the most common two categories of shapes to be created; block and cylinder. These shapes are always static-typed and can be utilized to construct features of city-like scene, e.g. buildings in Figure 57.

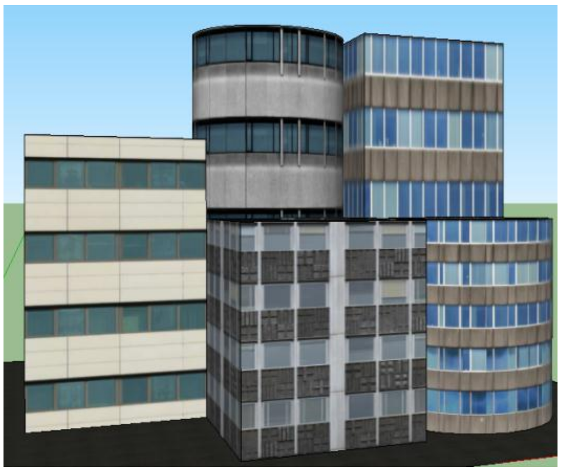


Figure 57. Buildings Created from Blocks and Cylinders

The process of block and cylinder generation is similar to the model import. The programmer only needs to specify required parameters in the constructor including 2D position of the shape and then its Z position will be calculated by the API. Shapes can also be elevated by giving positive Z offset as well.

6.2.1 Block

There are three steps of block generation. First is to create Block instance where the programmer can define name, position, z offset, width, length, height, and rotation of the block. Next step is to call the API method for adding shape to the 3D scene and the last step is to call Terrain method's texture(), in Appendix B.4, which requires texture of block's side and upper face as arguments before sending Ruby commands for block texturing to SketchUp. The upper face texture, however, can be left out of the arguments. If so, the API will apply a default black material as a roof texture. The example of block creation of the following code is exhibited in Figure 58.

```
Block b1 = new Block("b1", //name
              new Point2f(25,25),//position
              0,//Z offset
              50,//width
              50,//length
              100,//height
              0//rotation
           );
terrain.addShape(b1);
b1.texture("roof.jpg","bld1.jpg");
```

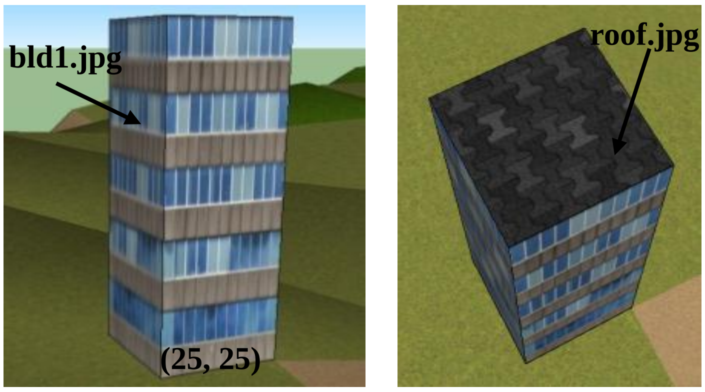


Figure 58. Block Generation

Block result, from Figure 58, is generated on the specific 2D position as defined. Z offset is set to zero so that the model doesn't float or sink to the terrain. A roof texture is applied on top of the block while side texture is applied on each side of the block as defined in the constructor.

6.2.2 Cylinder

Cylinder can be initiated similarly to block. The only difference in the constructor is that radius is needed, instead of width and height. The process of cylinder creation starts from initializing Cylinder instance, add it to the scene accordingly, and texture the shape. Figure 59 presents the example after cylinder is generated and textured.

```
Cylinder b2 = new Cylinder("b2",//name
      new Point2f(125,75),//position
      0,//Z offset
      25,//radius
      100);//height
terrain.addShape(b2);
b2.texture("roof.jpg","bld2.jpg");
```

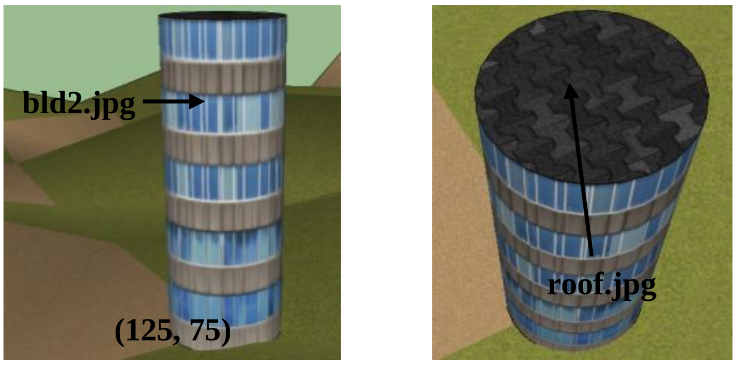


Figure 59. Cylinder Generation

The cylinder model is positioned and textured with different side and roof textures, accordingly to the defined parameters. It can be noticed that each model or shape in the scene requires its own constructor. It means that the programmer needs to initialize a new constructor each time a new scenery is created. This may cause a duplication problem if there are a big number of scenery models in the scene. In order to prevent the situation, the API provides another feature for loading and creating static scenery from a text file.

6.3 Statics Generation from Text File

The concept of static scenery generation from text file is to prevent exceed initiation of scenery constructors by allowing the programmer to define properties of each scenery model (or shape) as a single line in the file. Then the file is read and the information in each line will be extracted and utilized to generate the scenery. The scenery model and shape are defined differently in the file relatively to the following possible format.

b block-name x y width length height rotation texture-fileName [roof-texture-fnm]
c cylinder-name x y radius height rotation texture-fileName [roof-texture-fnm]
m fileName model-name x y scale rotation

The format of each type of scenery starts with specific letter indicating its type; block (b), cylinder (c), and model (m), followed by the arguments of each as described in the earlier sections. The following example presents content of a text file for creating scenery models and shapes in the previous sections.

```
//static tree
m tree.skp  m1 100 100 1 0
//block
b b1 25 25 50 50 100 0 bld1.jpg
//cylinder
c c1 125 75 25 100 0 bld1.jpg
```

The programmer can have the API read the file, extract the data and automatically generate the scenery on SketchUp side out of the file by calling Terrain's method loadStatics(). The following Java code demonstrates the behavior of method loadStatics() from Appendix B.4.

```
public void loadStatics(String fnm)
{
   //continuously read each line of the file
   String[] toks = line.split("\\s+");
   char staticCh = toks[0].toLowerCase().charAt(0);

   if(staticCh == 'b') {
      createBlock(toks);
   }else if (staticCh == 'c') {
      createCylinder(toks);
   }else if (staticCh == 'm') {
      createModel(toks);
   }else {
      //print error
   }
}//end of loadStatics()
```

The method processes each line in the file relatively to the first character of the line. Letter ‘b’ means than the line contains block information while ‘c’, and ‘m’ represents cylinder and static model information. Then, a method for creating static object out of information in the line is called accordingly.

7. USER MODEL AND CAMERA

User model is another part of the API components in Figure 22. It is the main dynamic model of the virtual world which can move and explore the scene. The model represents the user and can be followed by the camera, if preferred, unlike other dynamic models which cannot be followed by the camera. User model can be anything the programmer wants e.g. human, droid, and car. Figure 60 displays some of the possible user models that can be downloaded from SketchUp 3D Warehouse [30] and imported to the API.

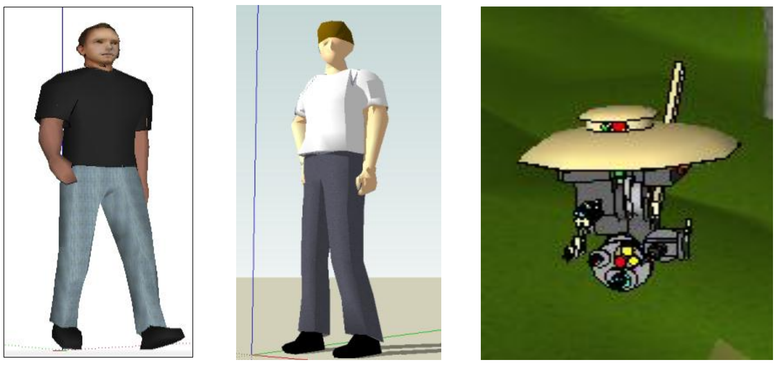


Figure 60. Some of the User Models

After the user model is added to the scene, another thing that becomes possible is creating a camera and set it to follow the model. Camera is another interesting feature of the API. It is responsible for manipulating the perspective of the scene. The API provides the programmer two options for managing the camera. It can either be set to look at certain position of the terrain (static camera) or attached to the user model (dynamic camera). Both types of cameras are demonstrated in Figure 61.

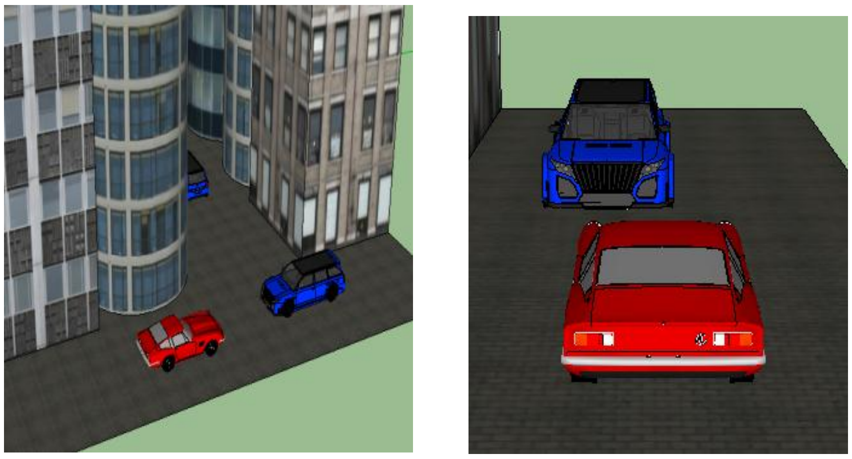


Figure 61. Static Camera and Dynamic Camera

Both kinds of cameras provide different perspective to the user. The static camera gives an overall look of the 3D scene while the dynamic is more real and gives closer view to the 3D scene features. This chapter describes how the user model is added to the scene and how to set the camera.

7.1 User Model

The user model is imported to the virtual world similarly to static and dynamic scenery models, only the user model instance is created from User class which contains the method for attaching the camera to the user model. Figure 62 shows the result of the user model generation from the example User constructor.

```
User user = new User("car.skp",//model file name
             new Point2f(170, 20), //position
             0, //offset Z
             0.2, //scale
             90); //rotation

terrain.addModel(user);
```

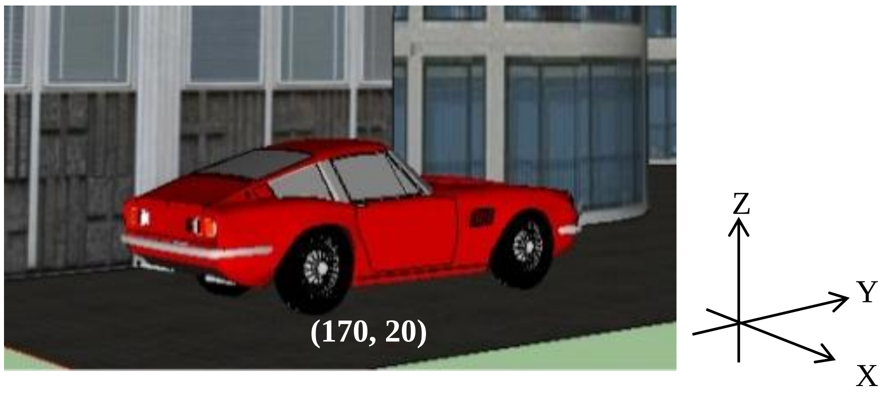


Figure 62. User Model Generation

The result of user model added to the scene, according to Figure 62, is the downsized red car with 90 degrees rotation. After the model is added to the scene, animation can be generated and assigned to the model. The user model is now available for the camera to be attached to it.

7.2 Camera

There are two main components of the camera; eye and target. The eye represents the position where the camera is set, while target is another point where the camera is staring at. The following subsection explains how eye and target are set differently in static and dynamic cameras.

7.2.1 Static Camera

In order to create a static camera, the programmer needs to define both the eye and target in format of Point3d and input the positions to the Scene method setCam() in Appendix B.14. Figure 63 displays an example showing where the eye and target of the static camera are after setCam() sends Ruby commands for static camera setting to SketchUp.

```
    Point3d eye = new Point3d(-10, 100, 406);
    Point3d target = new Point3d(60, 60, 168);
    terrain.setCam(eye, target);
```

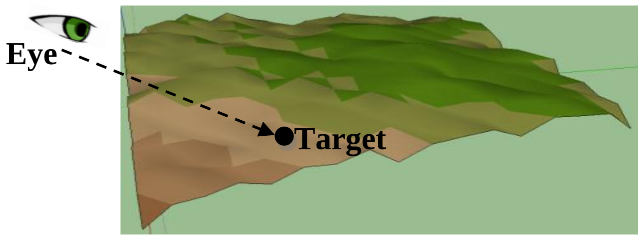


Figure 63. Cylinder Generation

The static camera creation in Figure 63 provides the overall perspective of the whole 3D virtual world. It is the best match in case the programmer wants to monitor behavior of every model in the scene at the same time.

7.2.2 Dynamic Camera

Camera can be set to follow the user model using setCam() in the User class in Appendix B.5, instead of Scene class. The method requires only camera's eye and automatically sets the camera target at the user model before assigning the model's steps array to the camera. Figure 64 indicates the example of camera attached to the user model.

```
    Point3d eye = new Point3d(170, 10, 50);
    user.setCam(eye);
```

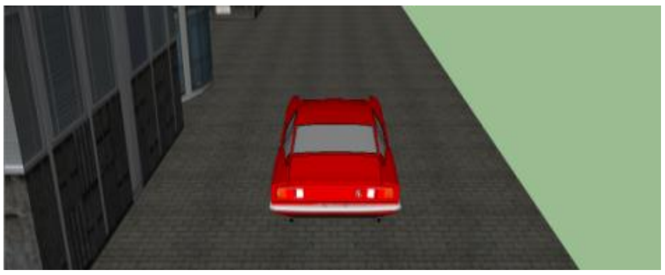


Figure 64. Camera Setting at the User Model

The camera following the user model is another option for the programmer. It gives a closer look to the scene and allows the users to explore the scene as if they were in it. Once the animation starts the camera moves along with the user model using the user model's animation. The process of generating the animation in terms of sequential steps arrays is described in more detail in the next chapter.

8. ANIMATION

Animation is the process for generating motion for each dynamic model as a sequence of small and continuous movements. The animation provided by the API in Figure 22 is discrete-event-based, which means the status and animation of each model could be altered during the run-time e.g. stop or change the movement direction of dynamic model. The animation model showing an interaction between animation in the 3D scene and Java program is presented in Figure 65.

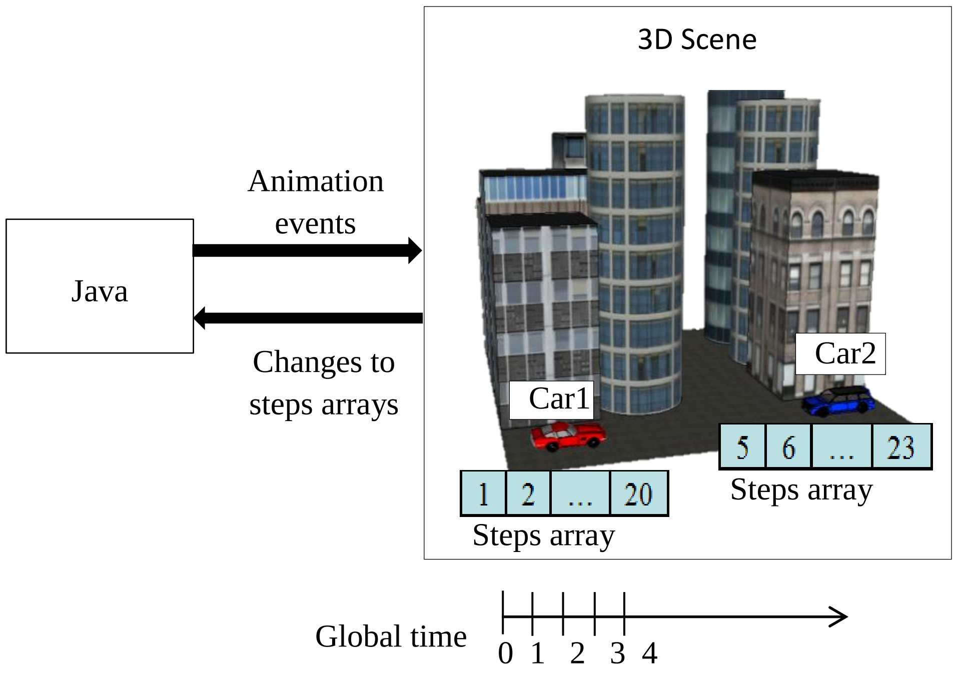


Figure 65. Animation Model

In the scene, each dynamic model is assigned with the steps array determining the motion, start time, and stop time of its animation. The animation of each model begins and ends accordingly to the system global time, for instance, Car1 starts moving at time one while Car starts at time five. The API defines a single time unit as one second and the permission for the programmer to modify that is not allowed. After the models animate, if there is any animation-related event occurs e.g. the beginning of each model's animation, and the collision between models, Java will detect and fire a method for handling each event as the programmer defined e.g. modify the steps array of models to recover the collision. There are five issues of the animation model; steps array generation, animating model with steps array, collision detection, event listener, and collision recovery. This chapter issues the first two stages of the animation.

8.1 Steps Array Generation

The process of movement generation produces the steps array as a result. In each element of the array, the position and orientation of the model at specific time unit are generated and stored. The example of animation from steps array is presented in Figure 66.

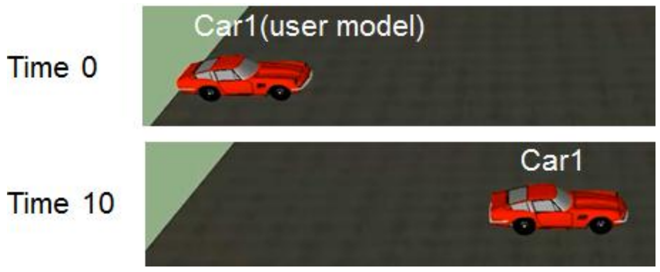


Figure 66. Animation Example of Car1

The animation example shows the closer view of the user model Car1 animating for a certain distance during the time unit 0-10. The model's steps array starts at time unit 1and ends at 20, according to Figure 66. The programmer can generate steps array for each model utilizing the method genStepsArray() in Terrain class from Appendix B.4.

```
Step[] steps1 = terrain.genStepsArray(
                  new Point2f(10, 100),//start position
                  new Point2f(180, 100),//stop position
                  0,//start rotation(optional)
                  0,//stop rotation(optional)
                  1,//start time
                  20//stop time
                );
```

The programmer can adjust start, stop position, start angle, stop angle, start time and stop time of the animation. Then the method genStepsArray() generates an array of Step instance containing position and rotation of the model at specific time stamp. The visual view of the animation of Car1 crossing the terrain is stated in Figure 67.

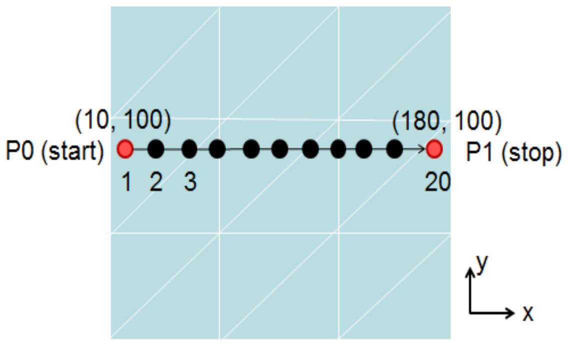


Figure 67. Example of Car1 Movements

It can be noticed that the programmer only defines the position of the model at the start and stop time in 2 dimension. In practical, in order to make the model animating in 3D scene, height Z for each coordinate is also required. Figure 68 shows a complete steps array after Z value of each coordinate are calculated and filled. Since the model Car1 is crossing a flat terrain, height Z of each movement is zero.

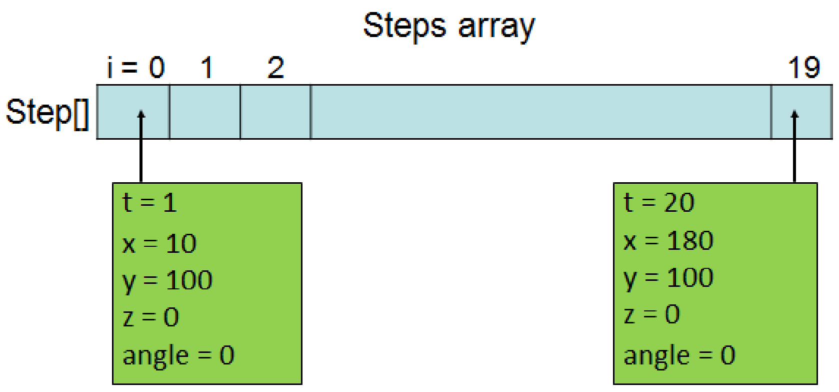


Figure 68. Example of Car1 Steps Array

According to the steps array in Figure 68, each element contains complete time stamp, x, y, z position, and angle of Car1 at the different time unit. All Coordinates Px, Py along the path from start to stop position are calculated using the following code fragment in Terrain's method genStepsArray() from Appendix B.4, which is hidden in the API.

```
int numStep = stopTime - startTime;

float vx = (P1.x - P0.x);//X vector from start to stop position
float vy = (P1.y - P0.y);//Y vector from start to stop position
int angle = (sAngle - dAngle)/numStep;
float d = (Math.sqrt(Math.pow(vx, 2) + Math.pow(vy, 2)));
float interval = d/numStep;
for (int i = 0; i <= numStep ; i++) {
   dis = interval*i;
   px = (P0.x + vx * (dis / d));
   py = (P0.y + vy * (dis / d));
   z = terrain.getHeight(px, py);
   stepsArray[i] = new Step(startTime+i, px, py, z, angle);
}
```

The code prepares small vectors and rotation in X, and Y axes to the interesting coordinate and increment the X, Y value of the start position by the vector in order to find next coordinate in the path in each iteration of the loop. Then, the height of each coordinate is calculated by the API using Terrain method getHeight() implemented employing Barycentric approach [29] in Figure 69.

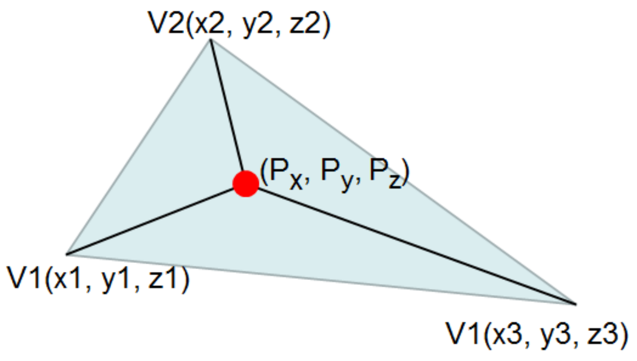


Figure 69. Example of Car1 Steps Array

Assuming that each coordinate in the animation path is a member of terrain triangle, what getHeight() does is to first specify which polygon, in the terrain, the given coordinate is in. If the polygon is found, any position in the triangle is represented as the weighted average of the triangle's vertices. If P_x and P_y are known, P_z can be calculated by the following equations in method getHeight().

$$\det_{xy}(v_1,v_2,v_3) = (y_1 - y_3)(x_2 - x_3) + (y_2 - y_3)(x_3 - x_1)$$
$$b_1 = [(P_y - y_3)(x_2 - x_3) + (y_2 - y_3)(x_3 - P_x)] / \det_{xy}(v_1,v_2,v_3)$$
$$b_2 = [(P_y - y_1)(x_3 - x_1) + (y_3 - y_1)(x_1 - P_x)] / \det_{xy}(v_1,v_2,v_3)$$
$$b_3 = [(P_y - y_2)(x_1 - x_2) + (y_1 - y_2)(x_2 - P_x)] / \det_{xy}(v_1,v_2,v_3)$$
$$P_z = b_1 z_1 + b_2 z_2 + b_3 z_3$$

However, chances are that the interesting coordinate in the path and the next one might not be in the same polygon. If the terrain is rough, the orientation of the polygons might not be in the same plane, as shown in Figure 70. The extra coordinate at the intersected edge of the polygons has to be calculated.

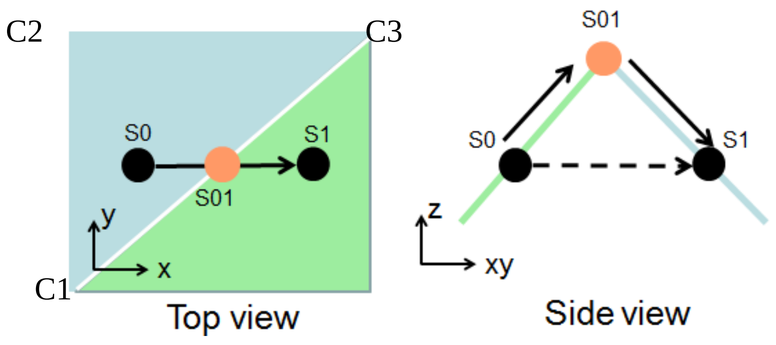


Figure 70. Extra Coordinate on the Intersected Edge

From Figure 70, if the object is moving from sub coordinate S0 to S1 which are in different polygons, S01 can be derived from two-line intersection approach [31]. In order to find the intersection point, each side of the polygon containing S0 will be passed to the intersection method, one will success (C1-C3), other two will fail (C2-C3 and C1-C2).

Suppose two lines are intersected, their vector are described as in Figure 71, the concept is to check first whether the lines are parallel using their slopes, if they are not, *t* and *u* will be solved and used to determine whether the lines are intersected and calculate the intersection point as well [31].

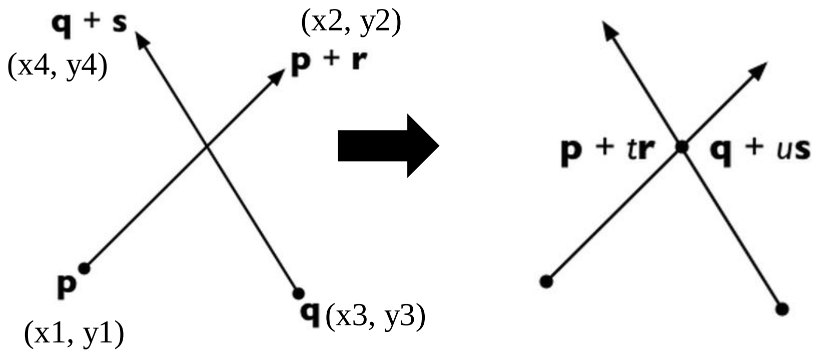


Figure 71. Linear equations for two intersected lines

Java code for intersection point calculation in method calcIntersection() from Appendix B.9,

```
double denom = (y4 - y3)*(x2 - x1)-(x4 - x3)*(y2 - y1);
if (denom == 0.0){ // Lines are parallel.
    return 0;
}

double ua = ((x4-x3)*(y1-y3)-(y4-y3)*(x1 - x3))/denom; //t
double ub = ((x2-x1)*(y1-y3)-(y2-y1)*(x1 - x3))/denom; //u

if (ua >= 0.0f && ua <= 1.0f && ub >= 0.0f && ub <= 1.0f) {
   // Get the intersection point.
   intersectP = new Point3f((x1 + ua*(x2 - x1)), (y1 + ua*(y2 -
                y1)),0);
    z = terrain.getHeight(intersectP.x, intersectP.y);
   intersectP.setZ(z);
}
```

The code solves ua and ub and uses them to calculate the intersection coordinate x and y. After x and y of the intersection point p is calculated, they will be used to search for the height h at that particular coordinate using method getHeight() and once the height is acquired, it is assigned to point p. Once the intersection points between polygons are prepared, the steps array in Figure 72 is updated in the loop in order to insert the intersection points. The Java code for this action is in Movement's method genStepsArray() in Appendix B.9 and the example after the insertion is in Figure 72.

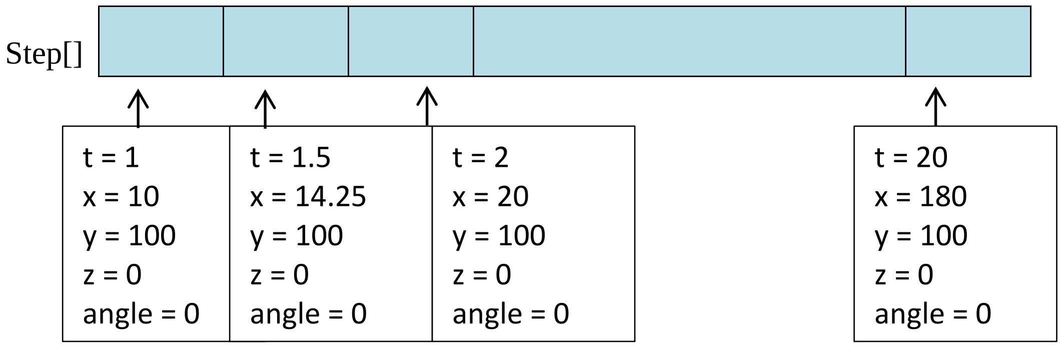


Figure 72. Steps Array after Transition Points Insertion

From Figure 72, it can be noticed that between the step at time one and time two, there is another step at time 1.5 inserted as a transition step. The time stamp of the transition step is always between the time of the previous and the next steps. Figure 73 shows the side view of the terrain demonstrating an example of animation after adding transition steps to the steps array.

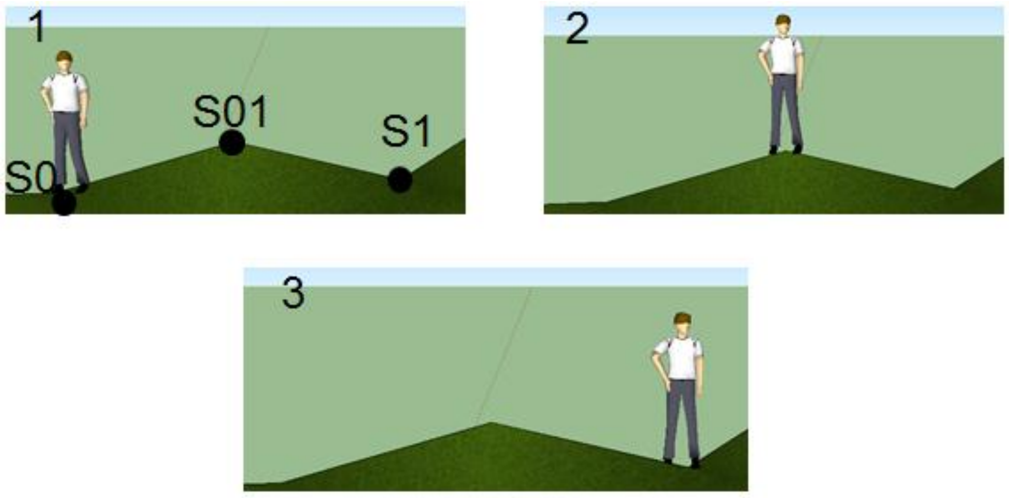


Figure 73. Result of Extra Coordinate Calculation

The result from picture one to picture three indicates that the model is moving up and down the hill instead of moving through. Once the steps arrays for movable models in the scene are prepared, they will be assigned to the model. The process of pairing the steps array with the model is explained in the next section.

8.2 Animating Model with Steps Array

The first step before the animation can start is to load the generated steps array in Java to the model in SketchUp. This is performed by calling the Model method setAnimation(), from Appendix B.2, of the model instance which writes the information in the array to the file and send Ruby commands to SketchUp for reading the file and creating animation of the model. The following code fragment from setAnimation() writes the information in steps array to the text file accordingly to the model name.

```
try (BufferedWriter out = new BufferedWriter(file)) {
  for (Step movStep1 : movStep) {
     out.write( movStep1.getTime() + ", " +
                                 (int) movStep1.getX() + ", " +
                                 (int) movStep1.getY() + ", " +
                                 (int) movStep1.getZ() + ", " +
                                 (int) movStep1.getAngle());
      out.newLine();
   }
   out.close();
 }
```

The code extracts the data in each element of a single steps array including time, 3D position, and angle, and then writes them to the file as a new line. However, if there are more than one steps arrays generated for the model, the API allows the arrays to be assigned to the model consecutively as an animation group. Assume that steps2 is the step array for animation during time 21-30, Figure 74 shows the animation group after steps1, from the previous section, is appended with steps2.

```
car1.setAnimation(steps1);
car1.setAnimation(steps2);
```

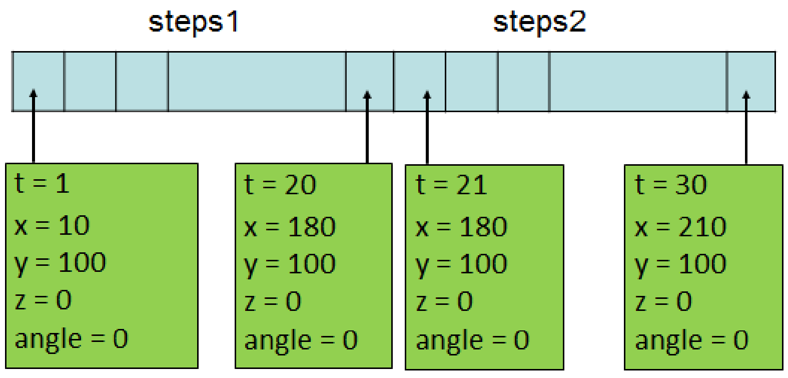


Figure 74. Car1 Steps Array after Being Appended

Since the new steps array of Car1 is composed of two steps arrays, it is possible that the start time of steps2 might not be continuous to the start time of step1. The API still append steps2 to steps1, as show in Figures 75.

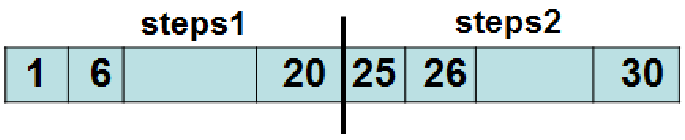


Figure 75. Steps Array with Waiting Time

For the animation result, Car1 will wait during the missing period between steps arrays. According to Figure 75, the model waits for 4 seconds during the time 21-24. Another problem is when the times of steps1 and steps2 overlap as shown in Figure 76.

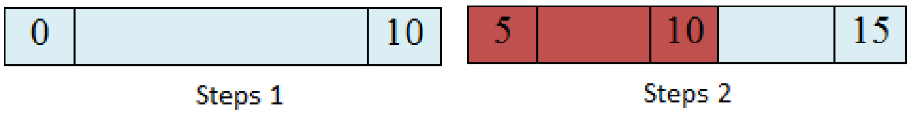


Figure 76. Overlapping Steps Arrays

The API always handles overlapped steps array by ignoring the overlapping period of the latest steps array, which is in steps2. For example, if steps2 starts at time 5, the API will get rid of the overlapping period of steps 2 from time 5-10 before appending the array. The process on SketchUp for reading step file, generating the animation group, and assigning the animation to the model occurs in method createAnimation(), in Ruby's Animation class, which is called by setAnimation() automatically after the steps array on Java side is written to the text file. The following code shows the activity in createAnimation() when each line is read from the file, stored in array 'line', and employed to create Step instance on the SketchUp side.

```
t0 = line[0].time
#create Step instance from line[0] and add to steps array
1.upto(line.length-1) {|i|

t = line[i].time
if t >t0 #ignore overlap time
    calculate deviation between previous and current lines
    generate Step instance and push to steps array
    t0 = t
else
    #'name' stored the model name
    msgFile.write("Warning: line #{i} in #{name}Step.txt is
    skipped\n")
end
}
msgFile.close()
```

According to the code for generating transformation array, time of the current line of step file is compared to the previous line. If the time t is greater than t0, the line will be read and utilized to create Step instance,

After the steps arrays are prepared and loaded to SketchUp's dynamic models, the animation can start by calling Terrain methods startAnimation(), from Appendix B.4, which sends the Ruby commands for initiating animation loop to SketchUp.

```
terrain.startAnimation();
```

The following pseudo code demonstrates the behavior of the animation processing loop implemented on SketchUp side.

```
time = 0
max = max stop time of models animation groups
(while time < max) {// global time of the scene
  for each model
    if time>= start time of the model
      if model's animation at time i exists
          animate model for 0.5 second
      else
          animate model for 0.5 sec using animation at time-0.5
      end
    end
  end
  time += 0.5
}
```

The global time of the animation loop is set to be incremented for 0.5 second at a time. As the global time of the scene is incrementing, the members of steps array at the current time of each dynamic model are processed. If the movement step representing animation of the current time exists, it will be used to animate the model for 0.5 second. The reason why the loop is incrementing global time for five seconds is to make it able to detect the transition steps in the array whose time are always multiply product of 0.5 second. If there is no transition step between the current ant next animations, for instance, no movement step at time 3.5 between current time three and time four, the movement steps of current time will be processed instead in this iteration (time = 3) and the next iteration of the loop(time = 3.5). This makes the model completely animated for one second from time three to four. As a result, each model continues animating accordingly to their steps array. During the animation run time, it is possible that the models might move past the same position and run through each other. This can be prevented by implementing collision detection. The next section describes the process of detecting collisions between these dynamic models and preventing them from moving through each other.

9. COLLISION DETECTION

During the animation, there is a possibility that there are intersection points between paths of dynamic models' animations which might cause them to run through each other. Such a situation decreases the realism of the animation and should be prevented. This chapter goes through the collision detection in the API, in Figure 22, which was designed and implemented for monitoring collisions between dynamic models and prevents them from passing through each other. Another purpose of detecting collision is to recover the collision so that the models can continue to their destination without crashing which is explained in the next chapter. The collision detection example is shown in Figure 77.

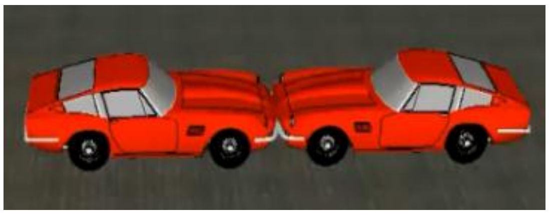


Figure 77. Collision between Dynamic Models

Figure 77 indicates that if the models stop before crashing after a collision is predicted by the API. There are two stages of collision detection in the process of generating 3D virtual world in Figure 78; dynamic collision detection and static. Static collision detection is performed first in the process of steps array generation while dynamic detection is performed during the runtime of animation.

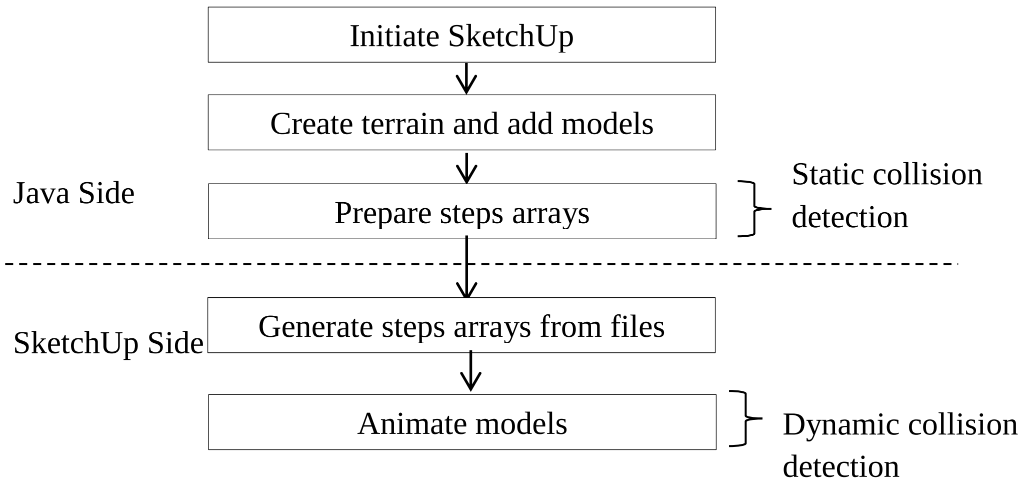

Initiate SketchUp
Create terrain and add models
Java Side
Prepare steps arrays
Static collision detection
Generate steps arrays from files
SketchUp Side
Animate models
Dynamic collision detection

Figure 78. The Process of Generating an Animated 3D World

The collision detection is performed twice during the process of generating an animated 3D world. The process starts from initiating SketchUp and Ruby console, followed by creating and texturing the terrain on which the models and shapes are placed. Then steps arrays for dynamic models are prepared and written in text files. Static collision detection is performed first in this stage in order to check whether each step in the array is within the terrain's boundary and does not fall in any of the existing static models' area. Then, on the SketchUp side, the files are read and steps arrays for each model are regenerated and assigned to each movable model. Then the models can be animated in the final stage which involves dynamic collision detection.

9.1 Static Collision Detection

This type of collision occurs between dynamic model and static model. Static collision is detected and prevented before the animation starts so it will not be seen during the animation runtime. Static collision detection is performed in steps array generation process on Java side reviewing in Figure 79.

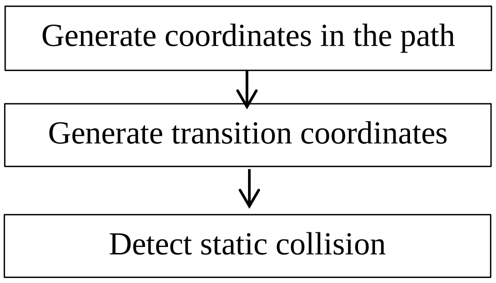

Generate coordinates in the path
Generate transition coordinates
Detect static collision

Figure 79. The Process of Steps Array Generation

During the steps array generation process, the static collision detection will be performed on each coordinate to specify whether it is off the terrain or within any of the static models' boundary. If there is a static collision, the API raises an exception to inform the programmer to change the destination and regenerate the animation. Figure 80 presents a diagram explaining the static collision detection approach.

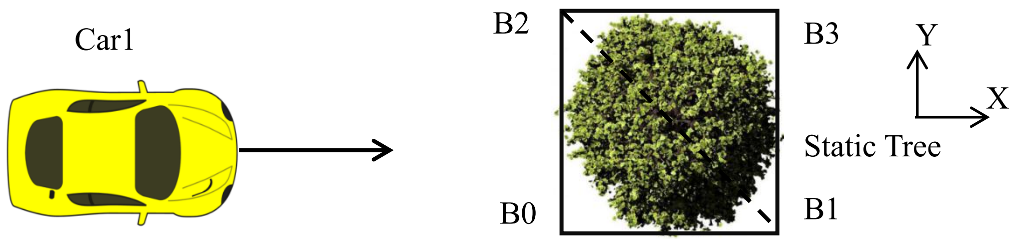


Figure 80. Static Collision Detection

Assuming that the steps array of Car1 means move it through a static tree, in order to detect whether each x, y coordinate in Car1's path is in the boundary of the tree or not, the static collision detection can be performed by dividing a boundary of the tree into two triangles; B0B1B2 and B1B2B3. Then the Barycentric approach is applied to each. If the coordinate happens to be in area of one of the triangles, the collision is detected. The static collision detection is implemented in method isCollision() in Movement class from Appendix B.9.

```
public boolean  isCollision(double x, double y){

  for each shape's boundary{
     if (x, y) is in triangle B0-B1-B2 or B1-B2-B3{
       return true;
     }
  }

  for each dynamic model's boundary{
     if (x, y) is in triangle B0B1B2 or B1B2B3{
       return true;
     }
  }
  return false;
}
```

The method validates if the interested coordinate is in the boundary of any shapes or static models composed of triangles B0B1B2 and B1B2B3. In order to do so, the coordinate and each vertex of triangle are utilized to calculate and verify accordingly to the Barycentric theory.

```
det = (v1.y - v3.y)*(v2.x - v3.x) + (v2.y - v3.y) * (v3.x - v1.x);
b1 = ((y - v3.y)*(v2.x - v3.x) + (v2.y - v3.y) * (v3.x - x)) / det;
b2 = ((y - v1.y)*(v3.x - v1.x) + (v3.y - v1.y) * (v1.x - x)) / det;

if ((b1 >= 0) && (b2 >= 0) && (b1 + b2 <= 1)) {
  //x, y is in triangle
  return true
}
```

9.2 Dynamic Collision Detection

Dynamic collision detection is utilized during the animation for avoiding dynamic models from moving through each other. The detection increases realism to the animation and is performed real-time before the model moves for each interval (0.1second) to the next position.

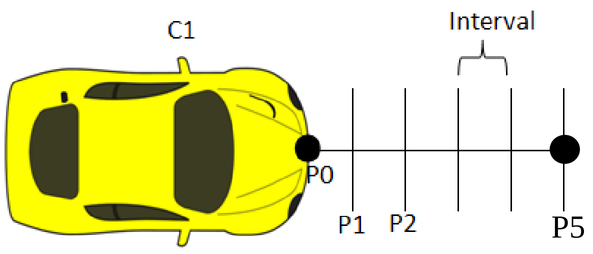


Figure 81. Dynamic Collision Detection before Each Interval

Figure 81 shows a car C1 moving for half a step to P5, containing five intervals. Before Car1 starts moving, the collision detection is performed at position P1 first. Collision detection is performed before the model moves for each small interval in order to keep detecting the collision as frequently as possible. The following Ruby code shows an animation loop, from section 7.2, after being updated with dynamic collision detection.

```
time = 0
max = max stop time of models animation groups
(while time < max) {# global time of the scene
  for each dynamic model
    if time>= start time of the model
      if model’s animation at time i exists
          0.upto(4){#animate for five intervals
            if !isCollision(nextX, nextY)
               animate model for 0.1 second
            end
          }
      else
          0.upto(4){#animate for five intervals
            if !isCollision(nextX, nextY)
               animate model for 0.5 sec using animation
               at time-0.5
            end
          }
      end
    end
  end
  time += 0.5
}
```

From the previous behavior of the loop, after the animation at specific time of the model is found, the model will be animated for 0.5 second (five intervals). The updated loop calculates a position of the next interval of the model relatively to the current position and the animation at current time and then verifies whether the position causes a collision with other dynamic models, instead of animating the model for five intervals straight. Figure 82 demonstrates the approach for detecting collision between dynamic models.

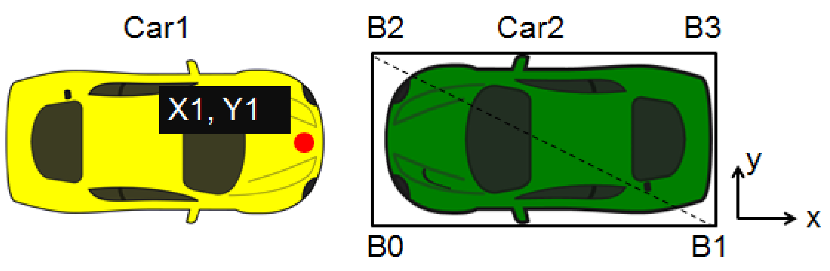


Figure 82. Dynamic Collision Detection Approach

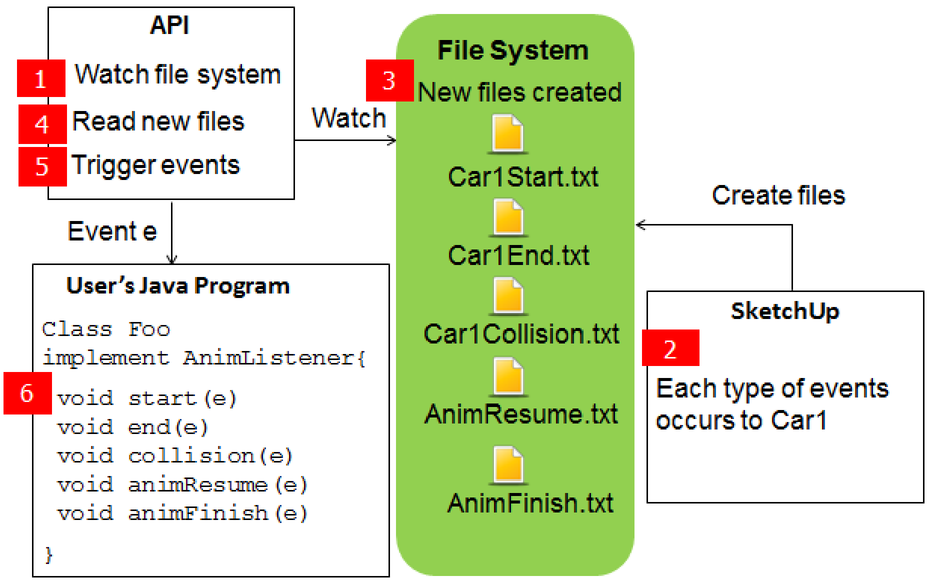
Assuming that Car1 is moving from x1, y1 to any coordinate x2, x2, the mechanism for dynamic collision detection between Car1 and another model Car2 is similar to static version. In order to detect collision between Car1 and Car2, the boundary of Car2 is split into two triangles. If x2, y2 falls in one of these triangles' region, the collision is detected. Dynamic collision detection is performed in method isCollision(), on SketchUp side.

```
def isCollision(x2, y2)
  for others dynamic models' boundary
    if x2, y2 is in triangle B1-B2-B0 and B1-B2-B3
      stop model
      create a new file '(modelName) + Collision.txt'
      file.write(collision data)
      return true
    end
  end
  return false
end
```

The method isCollision() validates the next position of the model. If the next position is in any of other dynamic models' area, the method temporarily stalls the model, creates a new file relatively to the model name, writes collision information to the file, and then returns true. Once there is a new collision file created and written in the project directory, a new collision event will be fired, and a method for automatically recover a collision by updating the steps arrays of collision models is called. The next chapter describes a mechanism of the API's event listener and event handler which also includes collision recovery.

10. ANIMATION LISTENER

The API supports discrete-event simulation in terms of event listener and handler. This allows the animation-related events to be detected and responded during the animation including start and end of each model's animation, collision, and resumption and end of all animations. This also allows the steps array of each dynamic model to be altered during the animation in order to prevent collision. After the animation starts, animation listener thread on Java side is initiated as shown in Figure 22. The listener monitors and responds to animation-related events. Figure 83 demonstrates six steps of event listener and handling.



```
Class Foo
implement AnimListener{
  void start(e)
  void end(e)
  void collision(e)
  void animResume(e)
  void animFinish(e)
}
```

Figure 83. Animation Listener Mechanism

The API employed a file system for SketchUp to send messages to Java. Due to the limit of SketchUp's API to other languages, the only interface to deliver information to Java, such as collision model and target name, is the file system. First on Java side after the animation begins, animation listener thread implemented using WatchService is initiated for monitoring the file system for any file creation and modification in step one. The thread is started in method Terrain's method startAnimation() in Appendix B.4. Then, on SketchUp side, a new file with unique name is created after each event is detected in step two and three, for instance, Car1End.txt is generated once the model Car1 reaches the end of its steps array. If the new file is detected, Java will use the file name to specify the type of event, read the file (step four), and create a new event object containing the information in the file in step five. Then in the sixth step, Java passes the event object to relative handling method defined by the programmer in Java program to response for the event, e.g. calling collision() to response to the collision event.

10.1 WatchService

After each event during the animation is caught, a new file will be created and written by SketchUp in Step 2 of Figure 83. The suitable approach to inform Java when each event is detected is to have Java monitor a specific directory in which the collision file will be created. Then create a new event relatively to the new file's name, for instance, if the new file's name contains 'collision', a collision event will be generated and Java will be informed that the collision is detected. In order to do so, a Java API for monitoring for changes in a specific directory is required. In this thesis, WatchService is utilized for implementing an animation monitoring thread which is responsible for performing step one, four, five, and six in Figure 83.

WatchService is a Java service that monitors the registered objects for the events that will occur to the objects e.g. changes to registered file or directory [7]. The process of monitoring directory using WatchService in animation listener thread is described in Figure 84.

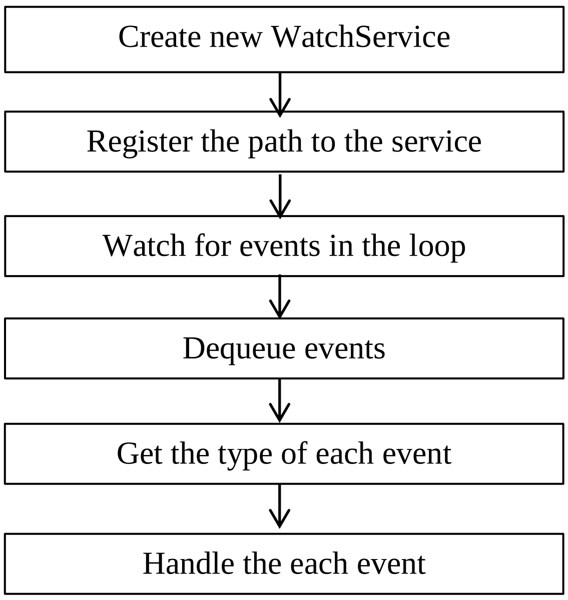

Create new WatchService
Register the path to the service
Watch for events in the loop
Dequeue events
Get the type of each event
Handle the each event

Figure 84. The process of using WatchService for monitoring the object's events

According to Figure 84, in order to monitor the events occurring in a particular directory, a new WatchService object is initiated first. Then the target directory is registered to the new service. In an infinite loop, registered directory are monitored for the incoming events such as creating, deleting, and modifying of files in the directory. If there is more than one event occurring in the directory, each event will be queued consecutively. The next stage is to dequeue the events, get the type of each event, and handle each type of event separately.

10.1.1 Create New WatchService

Animation listener thread takes the path of the directory as an argument from Terrain class. The code for each step of the process is in the method run() of Java class AnimThread, in Appendix B.13, which will be called automatically once the thread is started. A new WatchService instance is created by the following code.

```
// Obtain the file system of the Path
FileSystem fs = path.getFileSystem ();

// Create the new WatchService
WatchService service = fs.newWatchService();
```

The file system is created first out of the given directory path. Then it will be used to generate the new watch service. Once the service is generated, the file directory can be registered.

10.1.2 Register Path to the Service

A particular directory has to be registered to the service first, before it can be monitored. The interested types of events occurring to the object have to be given, as it is registered to the service. There are four kinds of events [32].

ENTRY_CREATE: Entry is created, or a new entry is moved or renamed.
ENTRY_DELETE: A folder/file is deleted, moved or renamed.
ENTRY_MODIFY: The contents of a file are modified.
OVERFLOW: Indicates that an event has been lost.

Example of the directory registration,

```
path.register(service, ENTRY_CREATE);
```

The java code above shows that the given path is registered to the service and the event needed to be watched is the creation of the files in the directory.

10.1.3 Watching and Handling Events

The rest of the process is described in this section, since they are in the same infinite loop. The following Java code in method run() watches for the events, dequeues the events, gets the kind, and handles each event inside the loop which are steps one, four, five, and six in Figure 83.

```
private AnimationListener lst;//user's program as listener
WatchKey key = null;
while(true) {
   //get the key to watch the events
   key = service.take();
   Kind<?> kind = null;
   // Dequeue and process each event
   for (WatchEvent<?> watchEvent : key.pollEvents()) {

     // Get the type of the event
     kind = watchEvent.kind();

     if(ENTRY_CREATE == kind){
        // A new Path was created
        Path evPath = ((WatchEvent<Path>) watchEvent).context();
        // Output
        String file = evPath.toFile().getName();
        if (file.contains("Collision")) {
           lst.collision(new CollisionEvent(this, file));
        }
        if (file.equals("AnimFinish.txt")) {
           lst.finish(new FinishEvent(this));
        }
        .
        .//handle other events accordingly
        .
     }
     if(!key.reset())//reset the key back to ready state
        break; //loop
   }
}
```

Once the directory is registered to the service, the event queue of the objects can be accessed using WatchKey instance. In each iteration, the new key is retrieved from the service. The event queue is extracted from the key and then each event in the queue is processed consecutively in a loop.

In each iteration of processing an event, the kind of each event is extracted. In this case, if the kind is ENTRY_CREATE, the file name that was created will be obtained and employed to create a new event. For example, if the name is 'AnimFinish.txt', it means the animation has now finished and if the file name contains 'collision', it means that the collision is detected. Each event object contains set of methods for getting the data written in the file. The following code fragment presents a structure of CollisionEvent class from Appendix B.11.

```
public class CollisionEvent {
   private String name, targetName;
   private int x, y, time;

   public CollisionEvent(Object source, String fileName) {
   //read information in the file and set to the variables
```

```
      //read step files of source and target
      //generate and store steps array of models from step files
      }
      public int getAngle()
      public String getSourceName()
      public String getTargetName()
      public Step[] getSourceStep()//get Steps array
      public Step[] getTargetStep()
      public Point2f getSourcePos()
      public Point2f getTargetPos()
      public Point2f getSourceDimen()
      public Point2f getTargetDimen()
      public int getTime()
   }
```

For example, after CollisionEvent is generated, the collision file will be read and the information e.g. collision model names, position, time, and rotation is stored in the object. The file names of the source and target of the collision are utilized to open their step files, generate the steps arrays of the models out of the files, and store them in the event object. After an event is identified and the event object is created, a Java handling method is called accordingly to the type of the detected event. The implementation of each method is in the user's program and its structure is described in the next section.

10.2 Event Handling Methods

In order to response to each fired event, AnimationListener is provided as an interface for event listener. The handling method of a listener is allowed to be defined in the Java program before the program itself is set as event listener by calling method setListener() in Appendix B.3. The following code fragment demonstrates a structure of the Java program written by the programmer in order to create a new animation listener and define event handling methods for specific event before start generating 3D virtual world.

```
   public class Foo implements AnimationListener{

      public void finish(FinishEvent e) {
         //code for handling finish event
         System.out.println("All animations has finished");
      }

     public void collision(CollisionEvent e){
        //code for handling collision event
     }
     .
     .
     .

     public static void main(String args[]){
        SketchUp skp = new SketchUp();
        skp.setListener(this);
        //code for generating 3D scene
        //code for starting animation
      }
    }
```

There are several handling methods in the main class e.g. collision() and finish(). These methods are available for the programmer to define. For example, the programmer can choose to exit SketchUp once the animations finish. Collision event is another important topic of the API. It is the only event that has a default response method basicRecovery() provided in the API. The programmer can simply pass an argument of collision() to this method and then collision is automatically solved. More detail about collision recovery and the default response method is in the next chapter.

11. COLLISION RECOVERY

Collision recovery is the process of fixing the collision between dynamic models during the animation. The recovery prevents the model from crashing to each other and allows the model to continue moving to the destination after a collision. Figure 85 presents a diagram demonstrating the collision recovery performed by the default collision responding method.

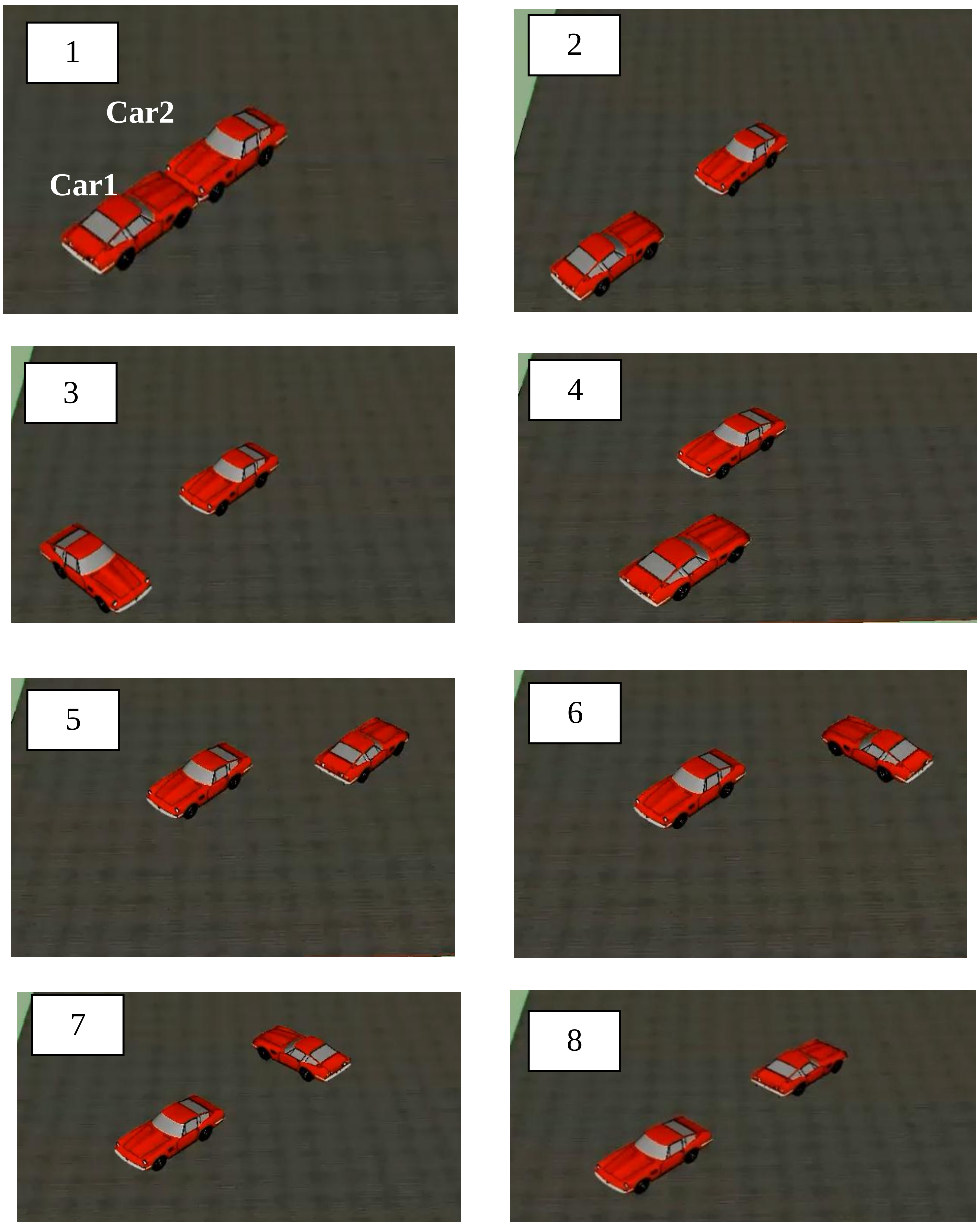


Figure 85. Default Collision Recovery

It can be noticed in Figure 85 that, after a collision, the model Car1 reverses back in picture number three and moves around Car2 by performing multiple rotations and translations, in order to avoid a collision. The API allows the programmers to either define their own methods for recovering collision in collision() or call the API's method basicRecovery() in CollisionRecovery class, in Appendix B.12, for solving the collision relatively to the example in Figure 85.

```
public static void basicRecovery(Terrain terrain, CollisionEvent e)
```

Utilization of basicRecovery() in Java program

```
public class Foo implements AnimationListener{

   public static void main(String args[]){
     //code for 3D world generation
   }
   public void finish(FinishEvent e) {
     System.out.println("All animations finished");
   }
   public void collision(CollisionEvent e) {
     CollisionRecovery.basicRecovery(terrain, e);
   }
   .
   .
   .
 }
```

The method basicRecovery() is called in collision() in the Java program. It requires terrain object and collision event instance as arguments. Figure 86 displays a diagram explaining the process of collision recovery, once the method is called.

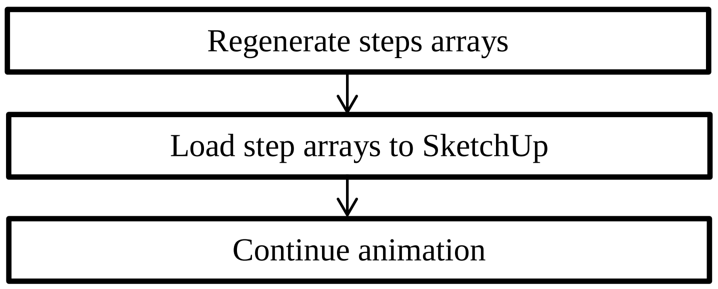


Figure 86. The Process of Collision Recovery

According to Figure 86, after the collision is detected, the collision event instance is generated. The instance contains the collision information such as model name, target name, collision time, and steps arrays of collision source and target. The steps arrays of both source and target are retrieved from the instance and utilized to generate new steps array. Then they are loaded to the source and target model on SketchUp side, before resuming the animation.

The steps array for both collision model and target are regenerated in order to let both models continue animating without crashing. The information from the collision event object e.g. width, height, time and position are employed for the regeneration. Figure 87 displays an paradigm for solving a collision between models in the method basicRecovery().

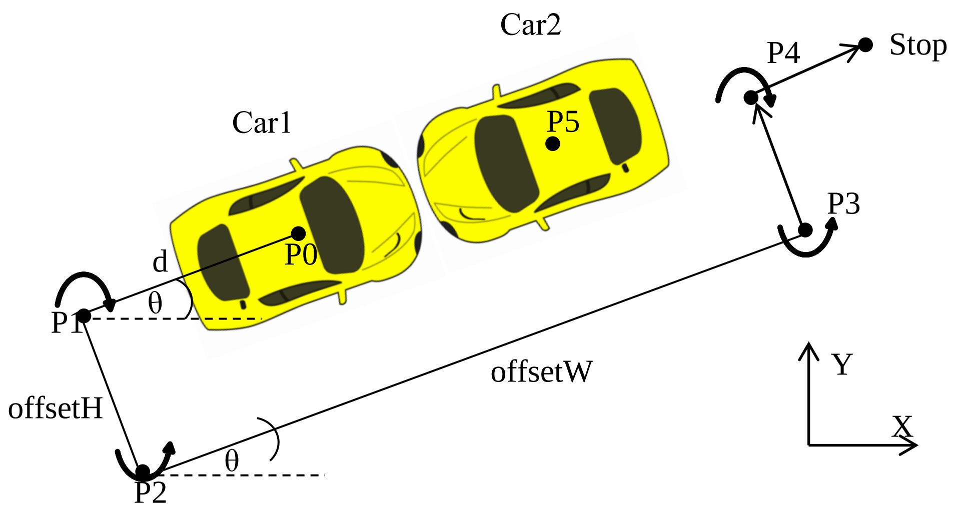


Figure 87. Collision Recovery Paradigm

The collision recovery approach in Figure 87 can fix collision in any angle. In order to solve a collision, Car1 reverses back and moves around Car2 from P0 to P4 and then continues its animation to stop position, while Car2 waits for three seconds before continues. The coordinates P1, P2, P3, and P4 are calculated accordingly to the collision models' position, dimension and angle. The angle of the model is stored in the model instance on the Ruby side, after the model is added to the scene. The angle is updated each time the model is animated and will be written to the file after a collision occurs. The utilization of models' position, dimension, and angle to calculate P1, P2, P3, and P4 can be described by the following equations.

d = Car1.width
offsetH = Car1.height + Car2.height
offsetW = Car1.width + Car2.width + d

P1 = (P0.x – d*cos(θ), P0.y – d*sin(θ))
P2 = (P1.x + offsetH*cos(90-θ), P1.y – offsetH*sin(90-θ))
P3 = (P2.x + offsetW*cos(θ), P2.y + offsetW*sin(θ))
P4 = (P1.x + offsetW*cos(θ), P2.1 + offsetW*sin(θ))

Since it is possible that the movement from P4 – stop might not be straightforward, it is necessary to keep the existing steps array from P4 – stop and update only the time stamp of each step. For example, assuming that the collision occurs at time 10, steps arrays of Car1 and Car2 are displayed in Figure 88.

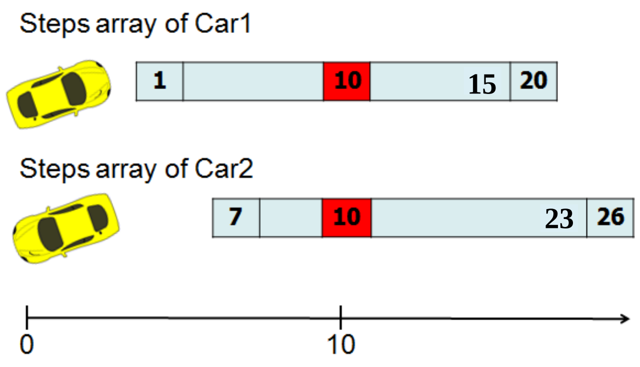


Figure 88. Steps Arrays of Car1 and Car2 at Collision Time

After a collision, the steps array for Car1 is updated so that the model moves around Car2. Assuming that the animation through points P0-P4 takes five seconds, the new array is generated as shown in Figure 89.

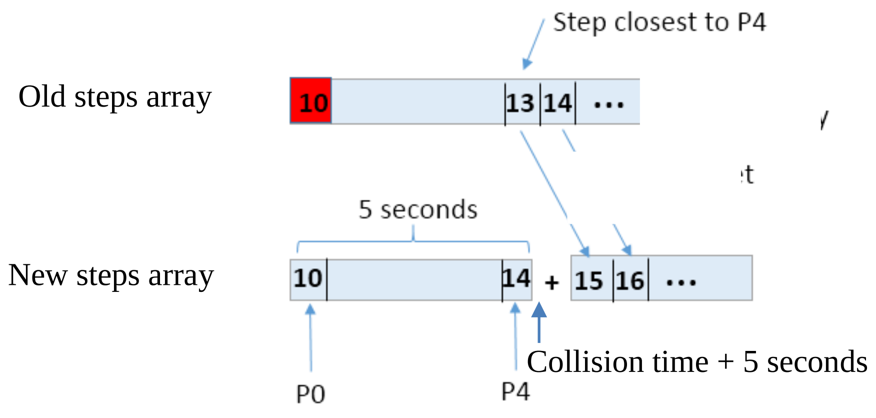


Figure 89. New Steps Array for Car1

The new steps array for Car1 is composed of steps array from P0-P4, which is generated by the API, and another subarray extracted from the existing steps array started at the position closest to P4. The start time of the first step of the subarray is assigned with the time after animation at P4 which is 15, according to Figure 89. Figure 90 describes the updated steps array of Car2 which waits for three seconds before continuing on its route.

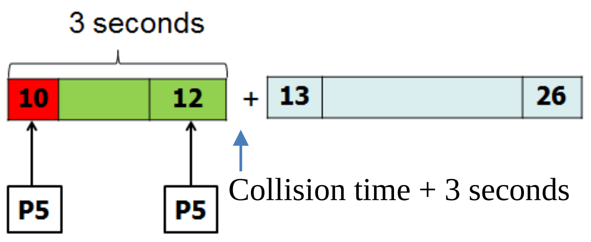


Figure 90. New Steps Array for Car2

The collision position of the Car2 (P5) remains unchanged from time 10 to 12, and then the old steps array sequence that started at time 11 is resumed at time 13, which is the time after Car finishes waiting. After the steps arrays of Car1 and Car2 are updated and reloaded, SketchUp will resume the animation starting from the collision time (time 10 in this example). The following code shows the behavior in basicRecovery() for creating new steps arrays for the collision models.

```
public static void basicRecovery(Terrain terrain, CollisionEvent e){

   /*sourceStep and targetStep receive steps array of source and
     target
  */
  Step[] sourceStep = e.getSourceStep();
  Step[] targetStep = e.getTargetStep);

  //calculate p0, p1, p2, p3, and p4 for collision source(Car1)
  //find the index of closest point to P4 in source step array:
  int p4Index = findClosestPoint(p4, sourceStep);

  //replace P4 by the closest position in the step file:
  p4 = new Point2f(sourceStep[p4Index].getX(), //x
                   sourceStep[p4Index].getY());//y

  //create steps arrays from P0-P4 taking 'offset' seconds
  //update time of source's steps array and get the array after P4:
  Step[] stepsAfterP4 = extractArray(p4Index+1, collisionTime +
                        offset, sourceStep);

  //get target(Car2) position at collision time
  Point2f targetPos = e.getTargetPosition();

  //find the index of target's position at collision time
  int targetIndex = findClosestPoint(targetPos, targetStep);

  //create steps array for target to wait for 'waitTime' seconds
  //update the time of target steps array:
  Step[] stepsAfterWating = extractArray(targetIndex+1,
                   collisionTime + waitTime, targetStep)

  //assigned new steps arrays to source and target
  //resume animation at collision time
}
```

The method starts by calculating P0, P1, P2, P3, and P4 for the collision source, then the x, y value of P4 are compared with the position of each step in the collision source steps array in order to find the index of the closest point to P4. P4 is replaced by the closest point and then the steps arrays representing the animation from P0 to new P4 are generated afterwards. The process continues by updating the time stamp of the collision source steps array after P4 using extractArray() which returns an updated array, started at a given time, as a result. For a collision target, the array is updated so that it starts at collisionTime + wait time. At the end, the steps arrays for both collision source and target are reloaded to the models on SketchUp side.

The following explanation is based on the regeneration of collision source steps array. After P4 is calculated, the closest position in the steps array is identified. Then P4 is replaced by the position, as shown in Figure 91.

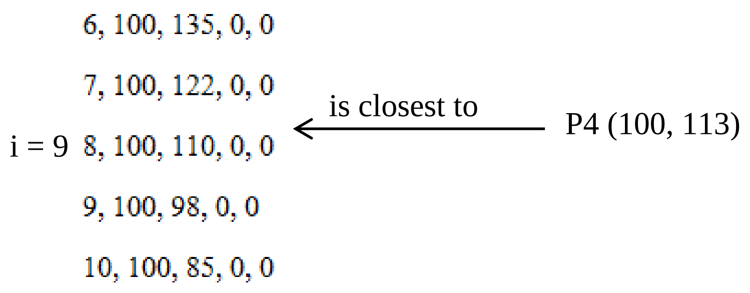


Figure 91. P4 is replaced by the closest point

The purpose of detecting the closest point to P4 is to find the index where the animation after P4 starts so that the steps array from P4 – stop can be updated correctly. According to Figure 91, subarray started at i=10 is updated. The following method in CollisionRecovery(), in Appendix B.12, describes the process of finding the index of the closest point to the calculated P4 by giving the x, and y position of P4.

```
public static int findClosestPoint(Point2f p4, Step[] stepsArray)
{
  Point2f p;
  int index = 0;
  Point2f start = new Point2f(stepsArray[0].getX(),
                              stepsArray[0].getY());
  float dist = start.distance(p4);
  for (int i = 1; i < stepsArray.length; i++) {
     p = new Point2f(stepsArray[i].getX(), stepsArray[i].getY());
     if (p.distance(p4)<= dist) {
        dist = p.distance(p4);
        index = i;
     }
     else{
        break;// if the distance gets bigger exit the loop
     }
  }
  return index;
}
```

After the index of P4 is found and the point is updated, the steps arrays representing the animation from P0 – P4 for Car1 and the stall for small amount of time of Car2 can be generated.

Source Step Arrays

```
int time = collisionTime
 //translate from P0 to P1
 sourceSteps1 = terrain.genStepsArray(p0, p1, 0, 0, time, time+1);
 //rotation at P1
 sourceSteps2 = terrain.genStepsArray(p1, p1, 0, -90, time+1,
      time+2);
 .
 .//translation and rotation at P2, P3, and P4
 .
 //updated steps array after P4
 stepsAfterP4 = extractArray(p4Index+1, time + offset,
      sourceSteps);
```

Target Step Arrays

```
//wait for two seconds
targetSteps = terrain.genStepsArray(p5, p5, 0, 0, time, time + 2);
//updated existing steps array
stepsAfterWaiting = extractArray(targetIndex, time + 3,
     targetStep);
```

There are nine steps arrays for the source including translation and rotation at P1, P2, P3, P4, and the steps array of the animation after P4. Two steps arrays are generated for the target including the stall for three seconds at P5 and the animation after that which is an time-updated existing steps array. The last steps arrays for source and target are extracted from their existing steps array by calling extractArray(). For example, extractArray() is called in order to process the existing steps array of Car1 started at p4Index+1, update the time stamp, create new Step instance out of each element, and create steps array after P4 for Car1.

```
public Step[] extractArray(int index, int time, Step[] stepsArray)
{

   Point2f p;
   int count = 0;
   String[] line;
   int t = time;
   int size = stepsArray.length - index;
   float x, y, z, angle;
   Step step;
   Step[] newSteps = new Step[size];

   //extract subArray started at index and given time
   for (int i = index; i < stepsArray.length; i++) {
      step = stepsArray[i];
      x = step.getX();
      y = step.getY();
      z = step.getZ();
      angle = step.getAngle();
```

```
            newSteps[count] = new Step(time + count, x, y, z, angle);
            count++;
        }
        return newSteps;
    }
```

The method generates a loop for processing the array of step file at the given index, which for the collision source is the index after where the position P4 is. The information in each line is extracted and the time stamp will be updated before a Step instance is generated from the line. The time of the first Step instance is set to the given time (collision time + time consumed by P0-P4, for the collision source) and will be incremented by 1 as the loop continues. At the end, the new step array representing the animation after P4 is returned. Now the steps arrays for P0-P4 and after P4 are prepared and ready to be reloaded to the collision source, before resuming the animation at the collision time. The steps array are set to the model by a loop in basicRecovery() which processes the array of dynamic models, compares the name of each model to the collision models name, and assign the steps array to the match accordingly.

```
for (Model model : terrain.getDynamicModels()) {
  try {
    if (model.getName().equals(e.getSourceName())) {
      model.setAnimation(mov1);
      model.setAnimation(mov2);
      .
      .
      .
      model.setAnimation(mov9);
    }
    if (model.getName().equals(e.getTargetName())) {
      model.setAnimation(tarMov1);
      model.setAnimation(tarMov2);
    }
  }
  catch (IOException ex)
  {//print error}
}
terrain.startAnimationAt(time);//resume animation
```

Figure 92 shows the steps arrays of the collision model and target after being updated and reloaded to SketchUp, the API will automatically resume the animation at the collision time by calling Terrain method startAmationAt().

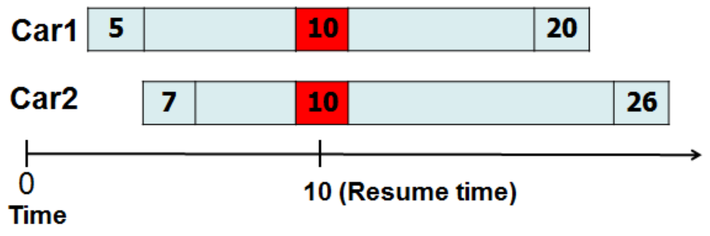


Figure 92. Updated Steps Arrays

Aside from findClosestPoint() and extractArray(), there are a number of other support methods provided to the programmer. Most of them are implemented for getting the information from collision file which are extracted and stored in CollisionEvent instance e.g. source name, target name, and collision time. The following list summarizes all support methods that are exposed to the programmer to use.

Support Methods in CollisionEvent Class

```
public String getSourceName()
public String getTargetName()
public Point2f getSourcePos() //x, y position
public Point2f getTargetPos()
public Point2f getSourceDimen() //(width, length)
public Point2f getTargetDimen()
public int getTime()
public float getSourceAngle()
public float getTargetAngle()
public Step[] getSourceStep()//get source steps array
public Step[] getTargetStep()
public Step getSourceStepAt(int time)//get step at specific time
public Step getTargetStepAt(int time)
```

Support Methods in CollisionRecovery Class

```
//return index of the closest point
public static int findClosestPoint(Point2f p4, Step[] stepsArray)

/*return subarray after a given index and update time of each
  element
*/
public static Step[] extractArray(int index, int time, Step[]
stepsArray)
//stop the model
public static boolean stopModel(String modelName)
```

The provided methods, along with movement generation method can cover various kinds of collision recovery. The concept of these methods is to provide the programmer access to the collision information as much as possible. Once the programmer can get to the information, any type of collision recovery can be designed and implemented as desired.

12. DISCUSSION

Several tests and measurements were performed on the API in order to indicate its advantages and weaknesses in different use cases, e.g. delay during the animation as there the number of dynamic models is increasing. This chapter discusses the API's limitations and also some feedbacks of the actual utilization of the API collected from the users.

12.1 API's Limitations

The limitations of the API are mostly from SketchUp specification and user's application. These factors occasionally cause the delay during the runtime of the API which slows down the animation.

12.1.1 Thread in SketchUp

SketchUp is the great tool which provides the most simplicity for 3D world creation among those 3D modeling tools available. However, the program still has its own downside; thread is not supported in SketchUp at all. Thread plays an essential role in each model's animation. If the animation the models can be processed separately in a thread, it is possible to animate the models simultaneously. However, since thread is excluded from the animation, the animation's processing loop is designed in order to provide the result closest to simultaneity.

```
time = 0
max = max stop time of models animation groups
(while time < max) {# global time of the scene
  0.upto(9){
    for each dynamic model

       if time>= start time of the model
         animate model at 'time' for 0.1 second
       end
    end
  }
  //increase time
}
```

The pseudo code for processing animation on SketchUp side shows a timing loop that consecutively animates each dynamic model for 0.1 second ten times in order to complete one second of global animation. As a result, SketchUp repeatedly switches between dynamic models and animates each one for a small distance. The total time elapsed during the runtime of five-second animation of a single model is collected three times and the result is displayed in Table 1.

Table 1. Actual Elapsed Time of Five-second Animation of a Single Model

Round	Elapsed Time (millisecond)	Delay (millisecond)
1	5153	153
2	5112	112
3	5197	197

Table 1 indicates that total animation time in each round slightly exceeds a defined animation time (five seconds). An average animation delay is 154 milliseconds which is 2.98 percent of the average elapsed time. It can be concluded that the animation loop does produce a slight delay during the animation of a single model, but it is small enough and will not cause big effect to the overall animation runtime.

However, according to the animation loop implementation, the delay tends to be higher if the programmer increases the number of dynamic models in the scene. Table 2 presents the five-second animation's elapsed time of three dynamic models.

Table 2. Elapsed Time of Five-second Animation of Three Models

Round	Elapsed Time (millisecond)	Delay (millisecond)
1	15909	10909
2	15985	10985
3	15943	10943

The table reveals that an average of animation delay rises to 10945.67 seconds after three dynamic models are used in the scene. This does affect the total runtime of the animation and can be noticed by the programmers. The cause of this big period of delay is the animation loop which cannot process the animation of each dynamic model at the same time, due to the limitations of SketchUp.

It can be concluded that the elapsed animation time tends to increase as the number of dynamic models gets higher. This problem is directly related to SketchUp's specification and does not seem to be solvable in the current version of SketchUp. The next section describes another software limitation of SketchUp which also cause the delay in the animation.

12.1.2 SketchUp's Interface to Java

Another disadvantage of using SketchUp is that there is no interface for SketchUp to communicate back to Java program. The communication is needed, when there is an animation-related event occurring on SketchUp side, so that it can inform the Java program and have Java deal with the event. To solve this problem, alternative interface is designed utilizing file system. The process of applying file system for handling the event is previously explained Animation Listener chapter. The event handler involves both file reading and writing which may cause a delay, for example, the delay in the collision recovery stage which is one of the main aims of the API. Table 3 shows the results of collision recovery delays during the process of solving a collision by basicRecovery().

Table 3. Delay from Collision Recovery

Round	Elapsed Time (millisecond)
1	9132
2	9107
3	9121

An average time of 9120 millisecond caused from a collision recovery is a long period of time which affects the total time of the animation. There are several factors that do not allow the problem to be solved. The collision recovery involves activities on both Java and SketchUp sides, as explained in Animation Listener chapter, including collision file creation on SketchUp, collision file reading and steps file manipulation on Java side in order to update the steps array, and steps file reading SketchUp side at the end. Aside from the delay from each step through the process, Autoit also causes some delays in delivering Ruby commands from Java to SketchUp to continue the animations as well.

These are the limitations of the API from SketchUp's disadvantages which cannot be gotten rid of or simplified unless the future version of SketchUp provide thread support and more optional interface to communicate with other programming languages.

12.2 User's Feedbacks

The API was tested by two programmers who graduated from the Department of Computer Engineering, Prince of Songkla University, and their opinions on the API were collected and discussed in this section. The tutorial of the API was shown and explained to each test subject, and then each individual was challenged to create the 3D scene on a rough terrain containing at least two dynamic models, one static model or shape, and animation.

The first test subject was the one who had some experience on manual utilization of SketchUp. As the tutorial went on, he noticed that the generation of a rough terrain produced the complex terrain result which seems to be slower and difficult if do it manually. He was also satisfied by the simplicity in animation creation. Only one question was asked, during the tutorial, which was where to download SketchUp models. His 3D scene is shown in Figure 93.

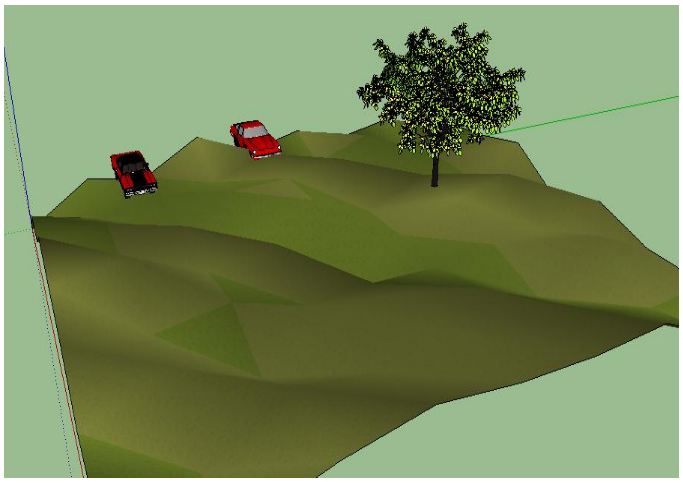


Figure 93. 3D Scene Created by the First User

The scene contains two dynamic cars, a single static three on a rough terrain as required. Each car translates along the surface of the terrain until it gets to the middle of the terrain. During the process, he claimed that it was slightly difficult to define each object position relatively to the size of the terrain, even in 2D. He created a mental model of the 2D terrain, separated them into four sections and then decided which model goes to where in order to predict the 2D coordinate. The user was also satisfied with the terrain multi-texturing and its result. Despite some complications, he thought that the API is easy to understand and apply and can finish the job that requires more time creating manually.

The other user was familiar with programming, but had no experience with SketchUp at all. She had a question about the logical order of the program and asked if it is possible to add some models without creating the terrain first, which is not allowed in the API. This user was interested in trying both flat and rough terrains and the scenes created by her is presented in Figure 94.

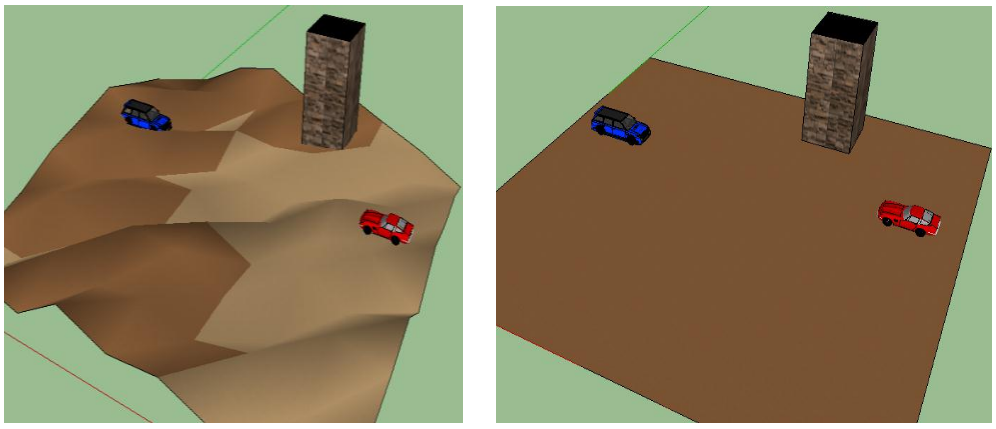


Figure 94. 3D Scenes Created by the Second User

The user was convinced to make something different from the one made by the first user using same provided resources. She created a desert-like scene with two cars and a big stone block and also decided to try collision recovery by basicRecovery(). Both cars in the scene translated into each other and collided at the middle of the terrain, before the collision recovery started. The user was pleased by the simplicity of the API, the 3D scene results and also the provided collision recovery, despite some noticeable delay during the process.

The practical usage of the API by the actual programmers indicates that the main aim of the API, its simplicity, was easily noticed by both users. However, despite the API's ease of use, the feedbacks prove that it still requires some familiarities and times in order to be able to use the API correctly e.g. to understand the programming order and limitations of the API.

Some changes can be made in the future version of the API based on the feedbacks of the users. The commands for printing a 2D mini map containing positions of objects in scene could be added so that the programmer can readjust each object's position without having to wait for the 3D scene to be completely generated. The programming order of the API could be improved in order to make each class more independent in order to get rid of the problem when the API requires a terrain instance before adding scenery.

13. SUMMARY

The existing 3D libraries in Java are designed for general needs and complicated for novice programmers to understand and apply. The main purpose of this thesis is to design and develop simple Java's 3D API for novices to generate 3D virtual worlds in SketchUp. The API provides features for creating components of the 3D scene including landscape generation, dynamic and static scenery models, user model, basic geometry creation, camera setting, and discrete-event-based animation. The work was summarized and the paper, included in Appendix D, was presented in The 7th PSU-UNS International Conference on Engineering and Technology. This chapter reviews the overview of the API, shown in Figure 95, and the API's features.

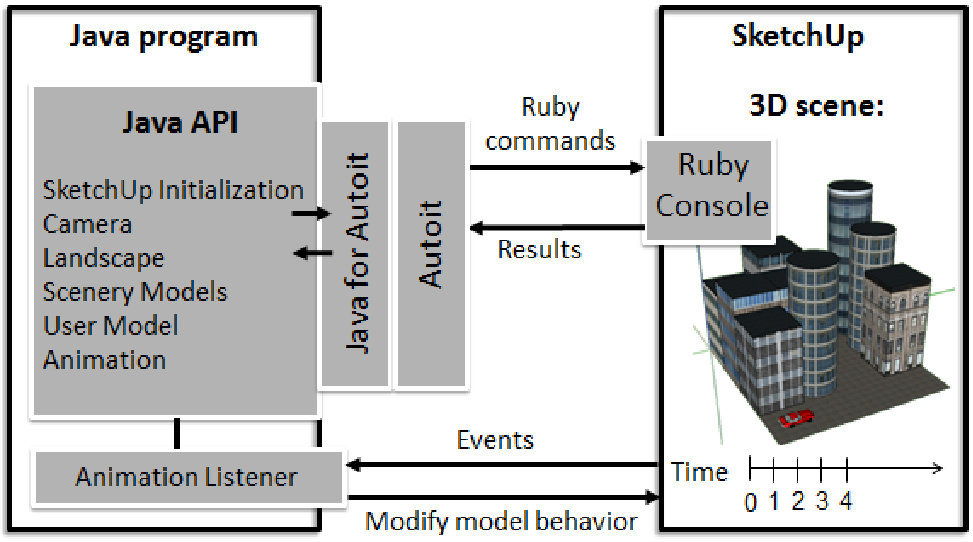


Figure 95. API overview

The API is utilized on the Java side inside the user's Java program. Each class of the API characterizes the appearance of 3D result that is on SketchUp side e.g. terrain model, user model, and animation. Once the user program is executed, SketchUp is initiated and Ruby commands from each class of the API are sent to Ruby Console in SketchUp utilizing Java interface for Autoit. The console executes and converts the input commands to the visual 3D results. Animation Listener thread on the Java side is started, after the animation on SketchUp side begins. The thread behaves as event listener and handler. If the event occurs during the animation, the thread will detect it and call the handling method defined by the user in the Java program which will modify the model behavior on the SketchUp.

13.1 SketchUp Initialization

SkethUp initialization is implemented using Java interface for Autoit (jAutoit). This feature is responsible for automatically initiating and creating new project in SketchUp, terminating SketchUp, and delivering Ruby commands from each class of the API to be executed on SkethUp. SketchUp is the main component of the API architecture. Once SketchUp window appears, ruby commands can be sent and executed from other classes of the API in order to start generating each part of the 3D world and initiating animation.

13.2 Landscape Generation

Landscape generation is responsible for characterizing the appearance of terrain which is a main component in the scene. The API provides three different ways of creating the terrain as Terrain class's constructors: terrain generation from Diamond-Square algorithm, coordinate file, and heightmap. Terrain generation from Diamond-Square algorithm creates unpredictable result of the terrain, however, the programmer can adjust whether the terrain is flat or rough and control the depth of the terrain. Terrain generation from coordinate file and heightmap creates a terrain relatively to the input file. These methods provide the programmer more control over the terrain result than the previous one.

Terrain class also contains features for texturing the terrain model. The textures can be divided into separate levels. The programmer can specify the number of texture levels and the material at each level. Then the API creates a transition between texture levels in order to make the terrain texture continuous.

13.3 Scenery Models

This feature creates both static and dynamic model representing the location of the scene. There are two types of scenery models available in the API; existing SketchUp models, and Geometric shapes. The existing SketchUp models can be downloaded and imported to the scene from SketchUp 3D Warehouse which contains variety of shared models. The programmer can adjust position, scale, rotation, and Z offset of the model and define whether it is static or dynamic (can be assigned with animation), before adding it to the scene by calling method addModel() in the Terrain class.

Basic geometry shape generation is also supported. The API contains constructor for creating block and cylinder which can be applied to build static scenery e.g. buildings. Similar to adding existing SketchUp model to the scene, the programmer can adjust 3D dimension of the shape, its position, scale, and offset in Z axis. Each shape can be textured by specifying the texture on top and side of it. The API also provides the method for creating static scenery from data in the file in case that there are a large number of static objects to be generated.

13.4 User Model and Camera

User model is the main dynamic model representing an avatar of the user in the virtual world. The user model can move and explore the scene and can be followed by the camera, if the programmer prefers. The user model is added to the scene similarly to static SketchUp models, only the user model instance is created from User class which contains the method for attaching the camera to the user model.

There are two ways to set up the camera in the scene. The first way (static camera generation) is to manually define the eye and target of the camera which are the position of the camera and the position where the camera is staring at. This creates a static camera which stares at a certain position. Another approach (dynamic camera generation) is to attach the camera to the user model. The programmer only needs to specify to eye of the camera. The camera's target is automatically set to the user's position and the camera follows the model once the animation starts.

13.5 Animation

The API provides discrete-event based animation which means the status of each model can be modified during the animation runtime. Animation of each dynamic model is represented in terms of steps array which is generated by calling method genStepsArray() in Terrain class. The programmer can adjust the start position and destination of animation, along with the rotation at both positions and the time period of the animation. The method calculates coordinate and rotation at each time stamp along the animation path and stores them in each element of the steps array. Then the steps array can be assigned to the dynamic model by calling setAnimation() in the Model class. Once the steps array for each dynamic model is set, the programmer can initiate animation by calling method startAnimaion().

The method startAnimation() triggers animation loop on SketchUp side. The loop repeatedly increments a global time of the animation and animates each model to the next position and orientation, at each time stamp, accordingly to its steps array.

13.6 Animation Listener

Animation listener is implemented utilizing WatchService as a thread for monitoring the animation-related events. The listener on Java side is initiated, after the animation starts. The listener monitors for events during the animation on SketchUp side i.e. start and end of each model's animation, and collision between models. The main concept is to have animation listener monitors for file creation event in the file system, which is the only interface for SketchUp to send messages to Java. A new file with unique name will be created and written by SketchUp, if there is the event occurs. If the listener detects a new file, it will read the file and create the event object relatively to the file name, for instance, CollisionEvent object will be created, if the file name contains 'collision'. Once the event instance is created, the listener triggers event handling method in user's Java program to response to the event accordingly to the kind of event.

13.7 Collision Detection and Recovery

Collision detection is the approach for predicting a collision between models in the API. Collision detection is grouped into two categories: static collision detection and dynamic collision detection. Static collision detection is performed when genStepsArray() is called in order to generate steps array. The static collision detection is performed on each coordinate in the animation path to verify whether the coordinate overlaps with any static scenery in the scene. If the static collision is detected, an exception will be raised. Dynamic collision detection, on the other hand, is performed during the animation runtime. Before each model is animated, the detection verifies whether there is a collision at the next position by checking if the coordinate of next position falls in any of the dynamic model's boundaries. Once the dynamic collision occurs, a collision file is created by SketchUp and collision event handling method is called by animation listener to perform a collision recovery.

Collision recovery is the process of fixing the dynamic collision. The API allows the programmers to either implement their own approaches of recovering collision utilizing support methods to access collision information or call the API method basicRecovery() which automatically updates collision models' steps array in order to solve a collision by having the collision target wait for small amount of time while the source is moving around it. As a result, each model can continue to its own destination without colliding with each other.

13.8 Conclusions

In Java, there are a number of 3D libraries available e.g. Java3D, JMonkey Engine, and JOGL. These libraries are capable of generating high-quality 3D result, but they are mostly designed to support general needs rather than aiming specifically at 3D world creation. Some of them are OpenGL based which makes it difficult for novice programmers to comprehend and apply. The main purpose of this research is to develop Java API for novice programmer focusing on the generation of 3D world in SketchUp including landscape, scenery models, user model, and camera setting. The API also supports discrete-event simulation in the form of animation, event listener, and collision recovery. The future work will focus on skeletal animation of the user model, better camera control, more complex shape generation, and background of the scene.

REFERENCES

[1] 3DS MAX, http://www.autodesk.com/products/3ds-max/overview, Accessed: 6/10/ 2015

[2] Unity3D, https://unity3d.com/unity, Accessed: 6/10/2015

[3] AutoCAD, http://www.autodesk.com/education/free-software/autocad, Accessed: 6/ 10/2015

[4] IMAGIS, http://imagis.pl/en, Accessed: 6/ 11/2015

[5] SketchUp, http://www.sketchup.com, Accessed: 6/10/ 2015

[6] Chopra, A. 2011. “Google SketchUp 8 for Dummies”. Indiana: Wiley Publishing, Inc., 1st ed.

[7] Oracle, http://www.oracle.com/technetwork/java/index.html, Accessed: 6/12/2015

[8] JOGL, http://jogamp.org/jogl/www/, Accessed: 5/21/2015

[9] jMonkeyEngine, http://jmonkeyengine.org, Accessed: 5/21/2015

[10] Autoit, https://www.autoitscript.com/site/autoit, Accessed: 6/10/2015

[11] Tal D. Learning to model terrain in SketchUp, http://www.sketchup-ur-space.com/2012/sept/learning-to-model-terrain-in-SketchUp.html, Accessed: 6/29/2015

[12] Schreyer A. 2012. “Architectural Design with SketchUp”. New Jersey: John Wiley & Sons, Inc., pp 1-4.

[13] Cleasby I. Trimble Sketchup Rendering, http://cleasby.co/sketchup_trimble/, Accessed: 6/29/2015

[14] Scarpino, M. 2010. “Automatic SketchUp: Creating 3-D Models in Ruby”. Hanover: Eclipse Engineering LLC, 1st ed.

[15] Ruby, https://www.ruby-lang.org/en, Accessed: 6/11/ 2015

[16] OGRE.RB, http://ogrerb.rubyforge.org, Accessed: 5/21/2015

[17] OGRE, http://www.ogre3d.org, Accessed: 5/21/2015

[18] G3DRuby, http://g3d-ruby.rubyforge.org, Accessed: 5/21/2015

[19] G3D Innovation Engine, http://g3d.sourceforge.net, Accessed: 5/21/2015

[20] OpenSceneGraph, http://www.openscenegraph.org, Accessed: 5/21/2015

[21] Irrlicht/Ruby Interface, http://irr.rubyforge.org, Accessed: 5/21/2015

[22] IRRLICHT, http://irrlicht.sourceforge.net/features, Accessed: 5/21/2015

[23] Seddon, C. 2005. "OpenGL Game Development". Texas: Wordware Publishing Inc.

[24] Learn OpenGL, http://learnopengl.com/#!Getting-started/Camera, Accessed: 9/16/2015

[25] Polack, T. 2003. "Focus on 3D Terrain Programing". Premiere Press, Ohio., pp 34-37.

[26] Landscape Terrain Using JOGL, https://azerdark.wordpress.com/2010/01/09/landscape-terrain-using-jogl/, Accessed: 9/16/2015

[27] Wright, R. et al. 2007. "OpenGL SuperBible". Pearson Education, Inc., Boston., 4th ed.

[28] OpenGL Tutorial, http://www.videotutorialsrock.com/opengl_tutorial/collision_detection/text.php, Accessed: 9/16/2015

[29] Dunn, F., Parberry, I. 2002. "3D Math Primer for Graphics and Game Development". Texas: Wordware Publishing Inc., pp 260-267.

[30] 3D Warehouse, https://3dwarehouse.sketchup.com/, Accessed: 6/29/2015

[31] Stackoverflow, http://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect, Accessed: 11/26/2014

[32] Ciobanu A. Java 7 Tutorial, http://andreinc.net/2013/12/06/java-7-nio-2-tutorial-writing-a-simple-filefolder-monitor-using-the-watch-service-api/, Accessed: 8/20/2014

APPENDICES

Appendix A. Java Code for API Examples

Appendix A includes Java code for some of the API examples; apartment room in page 3, house scene in page 2, and city scene in page 3.

Appendix A.1 ApartmentRoom.Java

```
/*
 *Create apartment room containing furniture,
  robots, and user model as shown in Figure 3
  (page 3) A collision is handled by making each
  model rotating right and moving straight for
  certain distance
 */
public class ApartmentRoom extends Scene implements
AnimationListener {

  private Terrain terrain;
  private Model crunchBot, cleaningBot, kiwiBot;


  public ApartmentRoom()
  {
    //initiate SketchUp and Ruby console
    SketchUp skp = new SketchUp();
    skp.setListener(this);
    skp.setDirectory("C:/");
    skp.startSketchUp();
    //construct 3D apartment room
    setCam(new Point3d(673, -20, 307), new
      Point3d(614, 6, 269));
    createTerrain();
    addWalls();
    addStaticModels();
    addDynamicModels();
    //create and start animation
    startAnimations();
}//end of ApartmentRoom()


private void createTerrain()
{
  /*
    Generate flat terrain(450*450) with single
    texture level, apply a floor texture, and
    enable shadow
  */
  try {
    terrain = new Terrain(0, 3, 50);
    terrain.setLevel(1);
    terrain.setTexture(1, "floor.jpg");
    terrain.texture();
    terrain.enableShadow();
  } catch (IOException ex) {
    System.err.println("IOException: " +
      ex.getMessage());
  }
}//end of createTerrain()

private void addStaticModels()
{
  /*
   Create and add static scenery models to the
   scene including sofa, bed, coffee table, tv,
   curtain, bookshelf, and picture.
   Require String file, Point2f p, int
   offsetZ(optional), float scale, float angle,
   boolean movable
  */
  Model sofa = new Model("sofa.skp", new
     Point2f(150, 150), 1,
   0,false);
  Model bed = new Model("bed.skp", new
     Point2f(100, 300), 1.2, 90, false);
  Model table = new Model("table.skp", new
     Point2f(150, 90), 1, 0, false);
  Model tv = new Model("tv.skp", new Point2f(150,
     20), 1, 180, false);

  Model curtain = new Model("curtain.skp", new
     Point2f(35, 380), 1, -90, false);

  Model bookShelf = new Model("shelf.skp", new
     Point2f(300, 390), 1, 0, false);

  Model picture = new Model("picture.skp", new
     Point2f(5, 200), 20, 1.5, 90, false);
  terrain.addModel(sofa);
  terrain.addModel(bed);
  terrain.addModel(table);
  terrain.addModel(tv);
  terrain.addModel(bookShelf);
  terrain.addModel(curtain);
  terrain.addModel(picture);
}//end of addStaticModels()


private void addDynamicModels()
{
  /*
   Add moving models on the room floor including
   crunchBot, cleaningBot, and kiwiBot(user model)
  */
  crunchBot = new Model("bigCrunch.skp", new
       Point2f(200, 80), 0.05, 45, true);
  cleaningBot = new Model("cleanBot.skp", new
       Point2f(300, 150), 0.3, 30, true);
  kiwiBot = new UserWrapper("kiwiBot.skp", new
       Point2f(200, 350), 3, 270);
  terrain.addModel(crunchBot);
  terrain.addModel(cleaningBot);
  terrain.addModel(kiwiBot);
}//end of addDynamicModels()


private void addWalls()
{
  /*
   Create flat blocks as room's walls. The block
   constructor requires 2D position, width,
   length, and height
  */
  Block wall1 = new Block(new Point2f(200, 400),
     400, 3, 100);
  Block wall2 = new Block(new Point2f(0, 200), 3,
     400, 100);
  terrain.addShape(wall1);
  terrain.addShape(wall2);
  wall1.texture("sky2.jpg");
  wall2.texture("black.jpg");
}//end of addWalls()


private void startAnimations()
{
  /*
   create steps arrays for dynamic models, disable
   shadow, and start animations
  */
  terrain.enableShadow();
```

```
try {
 Point2f p1, p2, p3, p4, p5, p6, p7;
 p1 = new Point2f(200, 350);
 p2 = new Point2f(200, 300);
 p3 = new Point2f(330, 170);
 p4 = new Point2f(200, 80);
 p5 = new Point2f(310, 190);
 p6 = new Point2f(300, 150);
 p7 = new Point2f(370, 150);

 //kiwiBot starts at p1, rotates at p2, and
 //ends and p3
 Step[] kiwiSteps1 = terrain.genStepsArray(p1,
       p2, 0, 3);
 Step[] kiwiSteps2 = terrain.genStepsArray(p2,
       p2, 0, 45, 3, 5);
 Step[] kiwiSteps3 = terrain.genStepsArray(p2,
       p3, 5, 13);
 kiwiBot.setAnimation(kiwiSteps1);
 kiwiBot.setAnimation(kiwiSteps2);
 kiwiBot.setAnimation(kiwiSteps3);

 //crunchBot moves straight from p4 to p5
 Step[] crunchBotSteps1 =
   terrain.genStepsArray(p4, p5, 0, 10);
 crunchBot.setAnimation(crunchBotSteps1);

 //cleanBot rotates at p6 and goes to p7
 Step[] cleanBotSteps1 =
   terrain.genStepsArray(p6, p6, 0, -30, 0, 3);
 Step[] cleanBotSteps2 =
   terrain.genStepsArray(p6, p7, 3,
 10);
 cleaningBot.setAnimation(cleanSteps1);
 cleaningBot.setAnimation(cleanSteps2);
 terrain.startAnimation();
```

```
  } catch (IOException ex) {
    System.err.println("IOException: " +
       ex.getMessage());
  }
}//end of startAnimations()


public void collision(CollisionEvent e)
{
  /*
   Extract collision time and then update the
   steps array of source and target by calling
   genSourceSteps() and genTargetSteps(), before
   resuming animations at collision time
  */
  int time = e.getTime();
  int offset = 20;
  genSourceSteps(e, offset);
  genTargetSteps(e, offset);
  terrain.startAnimationAt(time);
}//end of collision()


private void genTargetSteps(CollisionEvent e, int
offset)
{
 /*
   Extract collision target parameters (position,
   rotation, time, and name) and then call
   recoverCollision() in order to update its
   steps array
 */
 Point2f targetPosition = e.getTargetPos();
 float targetAngle = e.getTargetAngle();
 int time = e.getTime();
 String targetName = e.getTargetName();
 recoverCollision(targetName, time,
    targetPosition, targetAngle,
    offset);
}//end of genTargetSteps()


private void genSourceSteps(CollisionEvent e, int
offset)
{
  /*
    Extract collision source parameters (position,
    rotation, time, and name) and then call
    recoverCollision() in order to update its
    steps array
  */
  float angle = e.getSourceAngle();
  int time = e.getTime();
  Point2f position = e.getSourcePos();
  String sourceName = e.getSourceName();
  recoverCollision(sourceName, time, position,
    angle, offset);
}//end pf genSourceSteps()


private void recoverCollision(String modelName,
int time, Point2f position, float angle, int
offset)
{
  /*
   Update the steps array of the model relatively
   to the given model  name; the model turns right
   for 90 degrees (steps1) and move along for
   offset (steps2).
  */
  try {
    Step[] steps1, step2;
    Point2f p1;
    float x, y;
    x = position.x +
        offset*Math.cos(Math.toRadians(90 -
         angle));
    y = position.y - offset * Math.sin
        (Math.toRadians(90 - angle));
    p1 = new Point2f(x, y);
    steps1 = terrain.genStepsArray(position,
        position, 0, -90, time, time + 1);
    step2 = terrain.genStepsArray(position, p1,
        time + 1, time + 2);

    Model target = terrain.getModel(modelName);
    target.setAnimation(steps1);
    target.setAnimation(step2);
  } catch (IOException ex) {
    System.err.println("IOException: " +
        ex.getMessage());
  }
 }//end of recoverCollision()


 public void finish(FinishEvent e)
 {System.out.println("All animations finished");}


 public static void main(String args[])
 {new ApartmentRoom();}
}//end of ApartmentRoom
```

Appendix A.2 HouseScene.Java

```
/* Construct a scene containing static house, table,
   griller, trees, and dynamic cars as shown in
   Figure 1 (page No.2)
 */
public class HouseScene extends Scene implements
AnimationListener {

  private static Terrain terrain;
  private static Model redCar;
  private static Model yellowCar;

  public HouseScene()
  {
    SketchUp skp = new SketchUp();
    skp.setListener(this);
    skp.setDirectory("C:/ ");
    skp.startSketchUp();
    createTerrain();
    setCam(new Point3d(299, -383, 296), new
       Point3d(301, -315, 265));
    addStaticModels();
    addDynamicModels();
    startAnimations();
  }


  private static void createTerrain()
  {
    /*generate rough terrain with two texture levels
      from heightmap before applying street texture
      to the lower region and grass texture to the
      higher region
    */
     try {
    //create terrain
    terrain = new Terrain("C:/ heightmap.png", 10,
      10);
    terrain.setLevel(2);
    terrain.setTexture(1, "street2.jpg", 30);
    terrain.setTexture(2, "grass.jpg", 70);
    terrain.texture();
  } catch (IOException ex) {
    System.err.println("IOException: " +
       ex.getMessage());
  }
}//end of createTerrain()


private static void addStaticModels()
{
  /*add scenery models to the scene including a
    house, griller, trees, and table set
  */
  Model house = new Model("house.skp", new
      Point2f(320, 480), 8, 0.5, 0, false);
  Model griller = new Model("griller.skp", new
      Point2f(160, 320), 0, 1, 0, false);
  Model tableSet = new Model("tableSet.skp", new
      Point2f(320, 290), 0, 1, 90, false);
  Model tree = new Model("greenTree.skp", new
      Point2f(120, 480), 0, 0.5, 90, false);
  Model tree2 = new Model("greenTree.skp", new
      Point2f(520, 480), 0, 0.3, 90, false);
  terrain.addModel(house);
  terrain.addModel(griller);
  terrain.addModel(tableSet);
  terrain.addModel(tree);
  terrain.addModel(tree2);
}//end of addStaticModels()
private static void addDynamicModels()
```

```
{
  //add moving models including a yellow car and
  //another car
  redCar = new Model("redCar.skp", new Point2f(60,
      80), 0, 0.5, 0, true);
  yellowCar = new Model("yellowCar.skp", new
      Point2f(570, 60), 0, 0.5, 180, true);
  terrain.addModel(redCar);
  terrain.addModel(yellowCar);
}//end of addDynamicModels()


private static void startAnimations()
{
  Step[] yellowCarSteps =terrain.genStepsArray(
     new Point2f(570, 60), new Point2f(60, 60), 0,
     10);
  Step[] redCarSteps = terrain.genStepsArray(new
     Point2f(60,80), new Point2f(570, 80), 0, 10);
  redCar.setAnimation(redCarSteps);
  yellowCar.setAnimation(yellowCarSteps);
  terrain.startAnimation();
}//end of startAnimations()


private void genTargetSteps(CollisionEvent e, int
offset)
{
  /*get collision information and update steps
    array of the collision target
  */
  Point2f targetPosition = e.getTargetPos();
  float targetAngle = e.getTargetAngle();
  int time = e.getTime();
  String targetName = e.getTargetName();
  recoverCollision(targetName, time,
    targetPosition, targetAngle, offset);
  }//end of genTargetSteps()

  private void genSourceSteps(CollisionEvent e, int
offset)
  {
    /*get collision information of collision source
      and update the steps array the same way as
      collision target
    */
    float angle = e.getSourceAngle();
    int time = e.getTime();
    Point2f position = e.getSourcePos();
    String sourceName = e.getSourceName();
    recoverCollision(sourceName, time, position,
       angle, offset);
  }//end of genSourceSteps()

  private void recoverCollision(String modelName,
  int time, Point2f position, float angle, int
  offset)
  { /*update steps array of the model accordingly to
      a given name; the model rotate 180 degrees and
      continue for small distance
    */
    Step[] steps1, step2;
    Point2f p1;
    p1 = new Point2f(
         (position.x + offset * Math.cos
         (Math.toRadians(180 - angle))),
         (position.y - offset * Math.sin(
          Math.toRadians(180 - angle)))
         );
    steps1 = terrain.genStepsArray(position,
         position, 0, -180, time,time+ 1);
    step2 = terrain.genStepsArray(position, p1, time
        + 1, time + 5);
    Model target = terrain.getModel(modelName);
    target.setAnimation(steps1);
    target.setAnimation(step2);
  }//end of recoverCollision()

  public void finish(FinishEvent e)
  {System.out.println("finished");}

  public void collision(CollisionEvent e)
  { /*response to a collision by updating steps
      array of source and target before resuming
      animations
    */
    int time = e.getTime();
    int offset = 200;
    genSourceSteps(e, offset);
    genTargetSteps(e, offset);
    terrain.startAnimationAt(time);
  }//end of collision()

  public static void main(String args[])
  {new HouseScene();}

}
```

Appendix A.3 CityScene.Java

```
/**
 * Construct a city scene containing multiple shapes
   as buildings and some dynamic cars displayed in
   Figure 2 (page 3)*/

public class CityScene extends Scene implements
AnimationListener {

  private static Terrain terrain;
  private static Model redCar1, redCar2;
  private static Model yellowCar1, yellowCar2;

  public CityScene()
  {
    SketchUp skp = new SketchUp();
    skp.setListener(this);
    skp.setDirectory("C:/");
    skp.startSketchUp();
    createTerrain();
    setCam(new Point3d(261, -6, 715), new
       Point3d(261, 18, 624));
    addBuildings();
    addDynamicModels();
    startAnimations();
  }


  static void createTerrain()
  {   //generate flat terrain and apply street
      //texture on it
    try {
      terrain = new Terrain(0, 3, 50);
      terrain.setLevel(1);
      terrain.setTexture(1, "street2.jpg", 1);
      terrain.texture();
    } catch (IOException ex) {
      System.err.println("IOException: " +
         ex.getMessage());
    }
  }//end of createTerrain()
```

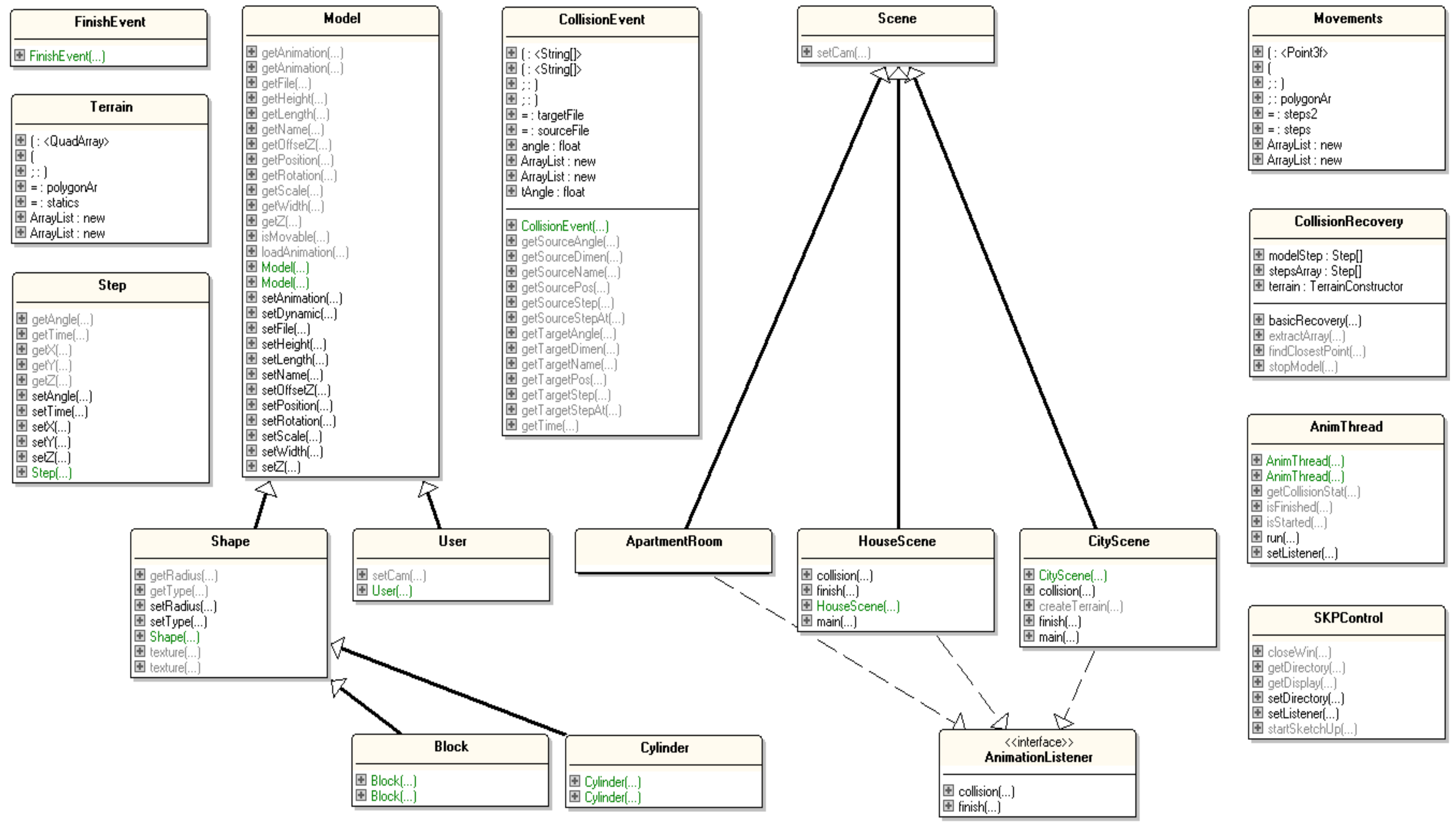
```
private static void addBuildings()
{
  /*create multiple blocks and cylinders as
    buildings from data in text file and add to
    terrain
  */
  terrain.loadStatics("building.txt");
}//end of addBuildings()


private static void addDynamicModels()
{
  /*add movable cars into the scene including two
    red cars and two yellow cars
  */
  redCar1 = new Model("redCar.skp", new
      Point2f(125, 75), 0.2, 0, true);
  redCar2 = new Model("redCar.skp", new
      Point2f(375, 375), 0.2, 270, true);
  yellowCar1 = new Model("yellowCar.skp", new
      Point2f(225, 275), 0, 0.25, 270, true);
  yellowCar2 = new Model("yellowCar.skp", new
      Point2f(375, 225), 0, 0.25, 270, true);
  terrain.addModel(redCar1);
  terrain.addModel(redCar2);
  terrain.addModel(yellowCar1);
  terrain.addModel(yellowCar2);
}//end of addDynamicModels()


private static void startAnimations()
{
  Step[] yellowCar1Steps = terrain.genStepsArray(
     new Point2f(225, 275), new Point2f(225, 25),
     0, 6);
  Step[] yellowCar2Steps = terrain.genStepsArray(
     new Point2f(375, 225), new Point2f(375, 25),
     0, 6);
  Step[] redCar1Steps = terrain.genStepsArray(new
     Point2f(125, 75), new Point2f(375, 75), 0,
     8);
  Step[] redCar2Steps = terrain.genStepsArray(new
     Point2f(375, 375), new Point2f(375, 25), 0,
     6);
  redCar1.setAnimation(redCar1Steps);
  redCar2.setAnimation(redCar2Steps);
  yellowCar1.setAnimation(yellowCar1Steps);
  yellowCar2.setAnimation(yellowCar2Steps);
  terrain.startAnimation();
}//end of startAnimations()


public void collision(CollisionEvent e)
{
  /*response to a collision by having souce model
    waits for two seconds before continue moving
  */
  int collisionTime = e.getTime();
  int waitTime = 2;
  genSourceSteps(e, waitTime);
  terrain.startAnimationAt(collisionTime);
}//end of collision()


private void genSourceSteps(CollisionEvent e, int
waitTime)
{
  Step[] sourceSteps1, sourceSteps2;
  Step[] oldSteps = e.getSourceStep();
  int index = 0;
  int time = e.getTime();
```

```
    Point2f p0 = e.getSourcePos();
    sourceSteps1 = terrain.genStepsArray(p0, p0,
    time, time + waitTime);
    index = CollisionRecovery.findClosestPoint(p0,
    oldSteps);
    sourceSteps2 = CollisionRecovery.
       extractArray(index, time +
       waitTime, oldSteps);
    Model source = terrain.getModel(
       e.getSourceName());
       source.setAnimation(sourceSteps1);
       source.setAnimation(sourceSteps2);
  }//end of genSourceSteps()


  public void finish(FinishEvent e)
  {System.out.println("finished");}


  public static void main(String args[])
  {new CityScene();}

}//end of CityScene
```

Appendix B. Java Code for API's Classes

Appendix B includes API class diagram and Java code for each class that can be utilized by the programmer.

Appendix B.1 Class Diagram



Appendix B.2 Model.Java

```
/*
 Model  class  contains  set  and  get  methods,  a
 constructor for creating Model instance and other
 methods for creating step file, writing steps array
 to  the  file,  and  creating  animation  on  SketchUp
 side
 */
public class Model {
  private static int count = 0;
  private String name;
  private Point2f position;
  private float height;
  private float angle;
  private float scale;
  private int width;
  private int length;
  private int sHeight;
  private int offsetZ;
  private String file;
  private boolean movable = false;
  private static SKPControl skp = new SKPControl();
  private static String SKP_TITLE = "Untitled -
     SketchUp";
  private static String EDITOR_TITLE = "Output -
     Project (run) - Editor";
  private static String RC_TITLE = "Ruby Console";
  private static JAutoIt lib = JAutoIt.INSTANCE;
  private Step[] stepsArray;


  public Model(String file, Point2f position,
  int offsetZ, float scale, float angle, boolean
  movable)
{
  /*
    Create static or dynamic model which is
    automatically named by the API If the model is
    dynamic, empty step file will be created
  */
  name = "Model" + count;
  count++;
  this.position = position;
  this.scale = scale;
  this.angle = angle;
  this.file = file;
  this.movable = movable;
  this.offsetZ = offsetZ;
  if(scale<=0){
    try {
      throw new ModelException("Model scale must
      be greater than zero");
    } catch (ModelException ex) {
      System.err.println("ModelException: " +
      ex.getMessage());
    }
  }
  //If dynmamic model, create an empty step file
  if (movable) {
    String path = skp.getDirectory() + name+
       "Step.txt";
    try {
      FileWriter modelFile = new FileWriter(path);
      BufferedWriter out = new
        BufferedWriter(modelFile);
      out.close();
    } catch (IOException ex) {
      System.err.println("IOException: " +
        ex.getMessage());
    }
    }
  }//end of Model()


  public Model(String file, Point2f position, float
  scale, float angle, boolean movable)
  {
    //Create Model instance without offsetZ
    this.name = "Model" + count;
    count++;
    this.position = position;
    this.scale = scale;
    this.angle = angle;
    this.file = file;
    this.movable = movable;
    this.offsetZ = 0;
    if(scale<=0){
      try {
        throw new ModelException("Model scale must
            be greater than zero");
      } catch (ModelException ex) {
        System.err.println("ModelException: " +
            ex.getMessage());
      }
    }
    //If dynamic, create step file for the model
    if (movable) {
      String path = skp.getDirectory() + name +
           "Step.txt";
      try {
        FileWriter modelFile = new FileWriter(path);
        BufferedWriter out = new
           BufferedWriter(modelFile);
        out.close();
      } catch (IOException ex) {
        System.err.println("IOException: " +
            ex.getMessage());
      }
    }
  }//end of Model()


  public void setName(String name)
  {this.name = name;}


  public String getName()
  {return name;}


  public void setPosition(Point2f position)
  {this.position = position;}


  public Point2f getPosition()
  {return position;}


  public void setZ(float height)
  {this.height = height;}


  public float getZ()
  {return height;}


  public void setRotation(float angle)
  {this.angle = angle;}


  public float getRotation()
  {return angle;}
```

```
public void setScale(float scale)
{this.scale = scale;}


public float getScale()
{return scale;}


public void setWidth(int width)
{this.width = width;}


public int getWidth()
{return width;}


public void setLength(int length)
{this.length = length;}


public int getLength()
{return length:}


public void setHeight(int height)
{sHeight = height;}


public int getHeight()
{return sHeight;}


public void setFile(String file)
{this.file = file;}
```

```
public String getFile()
{return file;}


public void setDynamic(boolean movable)
{this.movable = movable;}


public boolean isMovable()
{return movable;}


public void setOffsetZ(int offsetZ)
{this.offsetZ = offsetZ;}

public int getOffsetZ()
{return offsetZ;}


public void setAnimation(Step[] stepsArray)
{
  /*
    Read incoming steps array and write each
    element to step file of the model as a new
    line: time, x, y, z, rotation
  */
  if (stepsArray == null){
   try {
     throw new MovementException("Empty steps
        array is found");
   } catch (MovementException ex) {
     System.err.println("MovementException: " +
        ex.getMessage());
   }
```

```
    }
    this.stepsArray = stepsArray;
    String path = skp.getDirectory()+ name +
      "Step.txt";
    try {
      FileWriter file = new FileWriter(path, true);
      BufferedWriter out = new BufferedWriter(file);
      for (Step step : stepsArray) {
        out.write(step.getTime() + ", " +
        (int) step.getX() + ", " +
        (int) step.getY() + ", " +
        (int) step.getZ() + ", " +
        (int) step.getAngle());
        out.newLine();
      }
      out.close();
    } catch (IOException ex) {
      System.err.println("IOException: " +
         ex.getMessage());
    }
  }//end of setAnimation()


  public boolean loadAnimation()
  {
    //Send Ruby commands for creating animation for
    //the model to SketchUp
    lib.AU3_WinActivate(SKP_TITLE, "");
    lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
    lib.AU3_Send("anim.createAnimation("+
       name+"){ENTER}", 0);
    lib.AU3_Sleep(2000);
    skp.getDisplay();
    lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
       "");
    //Clear the display
    lib.AU3_WinActivate(EDITOR_TITLE, "");
    return true;
  }//end of loadAnimation()


  public Step getAnimation(int index)
  {return stepsArray[index];}

}//end of Model Class
```

Appendix B.3 SKPControl.Java

```
// SKPControl.java
// Andrew Davison, ad@fivedots.coe.psu.ac.th,
December 2013
 public class SketchUp {

  private static final String SKP_EXE = "C:/Program
     Files (x86)/SketchUp/SketchUp
     2013/SketchUp.exe";
  private static final String SKP_START_TITLE =
     "Welcome to SketchUp";
  private static final String START_BUTTON = "Start
     using SketchUp";
  private static final String SKP_TITLE = "Untitled
     - SketchUp";
  private static final String EDITOR_TITLE =
     "Output-terrain2(run) - Editor";
  private static final String RC_TITLE = "Ruby
     Console";
  private static JAutoIt lib = JAutoIt.INSTANCE;
  private static String directory;
  private AnimationListener lis;
```

```
/* If we get here then the Ruby Console is the
 * activewindow. It consists of two controls:
 * Edit2 is the display control (at the top)
 * Edit1 is the input text field (at the bottom)
 */
// enter commands into Edit1
public boolean startSketchUp()
 //Start Sketchup and press the start button to
 //get an empty window
{
  String skpPath = System.getenv("ProgramFiles") +
      "/" + SKP_EXE;
  if (!createWin(SKP_START_TITLE, SKP_EXE)) {
    System.out.println("Sketchup window could not
      be created");
    System.exit(1);
  }

  pressButton(SKP_START_TITLE, START_BUTTON);
  lib.AU3_Sleep(2000);

  if (lib.AU3_WinExists(SKP_TITLE, "") == 0) {
    System.out.println("\"" + SKP_TITLE + "\" does
       not exist");
    System.exit(1);
  }
  System.out.println("\"" + SKP_TITLE + "\"
       created");
  lib.AU3_WinActivate(EDITOR_TITLE, "");

  startRubyConsole();

  //here
  AnimThread msg = new
       AnimThread(Paths.get(directory));
  msg.setListener(lis);
  Thread msgThread = new Thread(msg, "message");
  //Start the threads

  msgThread.start();

  String terrainClass = directory + "Terrain.rb";
  String Shape = directory + "Shape.rb";
  String Model = directory + "model.rb";
  String Animation = directory + "Animation.rb";
  String cameraClass = directory + "Camera.rb";
  lib.AU3_WinActivate(SKP_TITLE, "");
  lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
  lib.AU3_Send("load{SPACE}\"" + terrainClass +
     "\"{ENTER}", 0);
  lib.AU3_Send("terrain = Terrain.new {ENTER}",
     0);
  lib.AU3_Send("load{SPACE}\"" + Shape +
     "\"{ENTER}", 0);
  lib.AU3_Send("shape = Shape.new {ENTER}", 0);
  lib.AU3_Send("load{SPACE}\"" + Model +
     "\"{ENTER}", 0);
  lib.AU3_Send("load{SPACE}\"" + cameraClass +
     "\"{ENTER}", 0);
  lib.AU3_Send("load{SPACE}\"" + Animation +
     "\"{ENTER}", 0);
  lib.AU3_Send("anim = Animation.new {ENTER}", 0);
  lib.AU3_Send("anim.setPath(\"" + directory +
     "\") {ENTER}", 0);

  return true;
}  // end of startSketchUp()

private boolean startRubyConsole()
 /* Assuming that the sketchup window is active,
    send alt-w alt-r to open the Ruby Console
    window
 */ {
  lib.AU3_WinActivate(SKP_TITLE, "");
  System.out.println("Bring up \"" + RC_TITLE +
     "\"");
  lib.AU3_Send("{^a", 0);
  lib.AU3_Send("{DEL}", 0);
  lib.AU3_Send("{ALT down}wr{ALT up}", 0);
  //alt-w, alt-r

  if (lib.AU3_WinWaitActive(RC_TITLE, "", 4) == 0)
  {
    System.out.println("\"" + RC_TITLE + "\" does
       not exist");
    System.exit(1);
  }

  System.out.println("\"" + RC_TITLE + "\"
    created");
  lib.AU3_WinActivate(EDITOR_TITLE, "");
  return true;
}  // end of startRubyConsole()

private boolean createWin(String title, String
  appName)
// run app and wait for specified window title
{
  if (lib.AU3_WinExists(title, "") != 0) {
    System.out.println("\"" + title + "\" already
       exists");
    lib.AU3_WinActivate(title, "");
    return true;
  }

  if (lib.AU3_Run(appName, "",
       JAutoIt.SW_SHOWNORMAL) == 0) {
    System.out.println("\"" + appName + "\" could
       not be executed");
    return false;
  }

  System.out.println("Waiting for \"" + appName
          + "\" window to appear...");
  lib.AU3_WinWaitActive(title, "", 10000);

  if (lib.AU3_WinExists(title, "") == 1) {
    System.out.println("\"" + title + "\" has
       appeared");
    return true;
  } else {
    System.out.println(appName + " has not
       appeared");
    return false;
  }
}  // end of createWin()

private void pressButton(String title, String
butStr)
{
  System.out.println("Pressing \"" + butStr + "\"
      in \"" + title + "\"");
  lib.AU3_ControlClick(title, "",
       "[CLASS:Button; TEXT:" + butStr + "]",
       "left", 1, JAutoIt.AU3_INTDEFAULT,
        JAutoIt.AU3_INTDEFAULT);
} // end of pressButton()

public boolean closeWin(String title)
// close window titled if it exists
{
  if (lib.AU3_WinExists(title, "") == 1) {
    lib.AU3_WinClose(title, "");
    if (lib.AU3_WinExists(title, "") != 0) {
        System.out.println("\"" + title + "\" not
          closed");
      } else {
        System.out.println("\"" + title + "\" has
          been closed");
      }
    } else {
      System.out.println("\"" + title + "\" does not
        exist");
    }
    return true;
  }  // end of closeWin()

  public String[] getDisplay()
  // return the Edit2 display text as an array
  {
    char[] display = new char[200];

    lib.AU3_ControlGetText(RC_TITLE, "", "Edit2",
            display, display.length);
    String displayText =
        AUUtils.chars2String(display);
    System.out.println("----------- Display --------
        ---");
    System.out.println(displayText);
    System.out.println("----------------------------
        ---");

    String[] lns = displayText.split("\\r?\\n");
    System.out.println("Display lines count: " +
        lns.length);

    return lns;
  }  // end of getDisplay()

  public void setDirectory(String directory)
  {this.directory = directory;}

  public String getDirectory()
  {return directory;}

  public void setListener(AnimationListener lis)
  {this.lis = lis;}

}  // end of SKPControl class
```

Appendix B.4 Terrain.Java

```
public class Terrain {

  private static int numInstance = 0;
  private static ArrayList<QuadArray> polygonAr =
        new ArrayList<QuadArray>();
  private static SKPControl skp = new SKPControl();
  private static final String SKP_TITLE = "Untitled
        - SketchUp";
  private static final String EDITOR_TITLE =
        "Output-terrain2(run)- Editor";
  private static final String RC_TITLE = "Ruby
        Console";
  private static JAutoIt lib = JAutoIt.INSTANCE;
  private ArrayList<StaticShape> statics = new
        ArrayList<StaticShape>();
  private String directory = skp.getDirectory();
  private String file = directory + "file.txt";
  private double[][] square;
  private int iter = 0;
  private int tSize = 0;
  private int sleep = 1;
  private int polygonSize = 0;
private ArrayList<Shape> shape = new
     ArrayList<Shape>();
private ArrayList<Model> models = new
     ArrayList<Model>();
private ArrayList<Model> movModels = new
     ArrayList<Model>();
private int level = 0;
private int texIndex = 0;
private String[] tex;
private int[] height;
private int depth = 0;
private float resolution = 0;


public Terrain(int depth, int resolution, int
plgSize)
{
  //generate terrain model from given resolution,
  //depth, and polygon size
  this.depth = depth;
  if (resolution <= 0) {
    try {
      throw new TerrainException("Terrain
          resolution must be greater than zero");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
          ex.getMessage());
    }
  }
  if (plgSize <= 0) {
    try {
      throw new TerrainException("Polygon size
         must be greater than zero");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
         ex.getMessage());
    }
  }

  if (numInstance < 1) {
    numInstance++;
    polygonSize = plgSize;
    tSize = resolution;
    this.resolution = (float) ((Math.pow(2, tSize)
        + 1) * polygonSize);
    //generate terrain coordinates
    initSquare((int) Math.pow(2, resolution) + 1);
    //create array of polygon
    createPolygon();
    try {
      saveTerrain(file, (int) (Math.pow(2, tSize)
         + 1));
    } catch (IOException ex) {
      System.err.println("IOException: " +
         ex.getMessage());
    }
    loadTerrain(file, plgSize);
  } else {
    try {
      throw new TerrainException("Cannot create
         multiple terrains.");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
         ex.getMessage());
    }
  }
}//end of Terrain constructor


public Terrain(String file, int plgSize)
 {
```

```
//geenerate terrain model from coordinate file
this.polygonSize = plgSize;

if (file.endsWith(".txt")) {
  //read coordinates from file
  try {
    BufferedReader br = null;
    try {
      br = new BufferedReader(new
         FileReader(file));
    } catch (FileNotFoundException ex) {
      System.err.println("FileNotFoundException:
         " + ex.getMessage());
    }
    String line;
    String[] info;
    int count = 0;
    int size, i, j, z;
    while ((line = br.readLine()) != null) {
      if (count > 0) {
        info = line.split(",");
        i = Integer.parseInt(info[0]);
        j = Integer.parseInt(info[1]);
        z = Integer.parseInt(info[2]);
        square[i][j] = z;
      } else {
        info = line.split(",");
        System.out.println(info[1]);
        size = Integer.parseInt(info[1]);
        tSize = size;
        square = new double[(int) Math.pow(2,
             size) + 1][(int) Math.pow(2, size)
             + 1];
      }
      count++;
    }
```

```
    } catch (IOException ex) {
      System.err.println("IOException: " +
          ex.getMessage());
    }
  } else {
    try {
      throw new TerrainException("Incompatible
          file type");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
          ex.getMessage());
    }
  }
  this.resolution = (float) ((Math.pow(2, tSize) +
          1) * polygonSize);
  //create array of polygons and create terrain on
  //SketchUp side
  createPolygon();
  loadTerrain(file, plgSize);
}//end of Terrain contructor


public Terrain(String file, int depth, int
plgSize) throws IOException
{
  //generate terrain model from heightmap
  this.polygonSize = plgSize;
  if (file.endsWith(".bmp")||
      file.endsWith(".png")||
      file.endsWith(".jpg"))
  {
    try {

      BufferedImage image = ImageIO.read(new
          File(file));
      square = new
      double[image.getWidth()][image.getHeight()];
      resolution = image.getHeight() * plgSize;
      for (int y = 0; y < image.getHeight(); y++)
      {
        for (int x = 0; x < image.getWidth(); x++)
        {
          /* in a grayscale map, the RGB parts
           will all be the same value,
           so just take one out of the packed ints
          */
          int height = depth - (int)
             ((image.getRGB(x, y) & 0xFF)
              / 255.0 * depth);
          if (height != 0) {
            System.out.printf("%01d ", height);
          } else {
            System.out.print("  ");
          }
          //store x, y, z in array of coordinate
          square[x][image.getHeight() - 1 - y] =
             height;

        }
      }
      saveTerrain(file, image.getWidth());
      System.out.println();
    } catch (IOException ex) {
      System.err.println("IOException: " +
            ex.getMessage());
    }
  } else {
    try {
      throw new TerrainException("Incompatible
          file type");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
          ex.getMessage());
    }
  }
  createPolygon();
  loadTerrain(file, plgSize);
}//end of Terrain constructor


private final void initSquare(int size)
{
  /*initialize height of each coordinate at each
    corner of terrain heightmap
  */
  square = new double[size][size];
  int k = size - 1;
  double t = Math.log(k) / Math.log(2);
  if (t != (int) t) {
    System.err.println("size must be of type 2^n +
       1");
    return;
  }
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      square[i][j] = 0;
    }
  }
  if (depth != 0) {
    square[0][0] = Math.random() * depth;
    square[0][k] = Math.random() * depth;
    square[k][0] = Math.random() * depth;
    square[k][k] = Math.random() * depth;
    //calculate height of other coordinates
    calcMids();
  }
  long stop = System.currentTimeMillis();
}//end of initSquare()


private void calcMids()
{
  //repeatedly calculate height of a midpoint out
  //of neighbor coordinates
  iter++;
  int offset = (int) ((square.length - 1) /
      Math.pow(2, iter));

  for (int i = offset; i < square.length; i += 2 *
  offset) {
    for (int j = offset; j < square.length; j += 2
    * offset) {
      square[i][j] =
               (square[i - offset][j - offset]
              + square[i + offset][j - offset]
              + square[i + offset][j + offset]
              + square[i - offset][j + offset]) /
              4;
      if (sleep != 0) {
        try {
          Thread.sleep(sleep);
        } catch (InterruptedException ex) {
        }
      }

    }
  }
  calcHalfs();
}//end of calcMids()


private double randomDisplacement()
{
  double t = (Math.random() / iter - 1 / (2 *
     iter)) * 64;
  return t;
}//end of randomDisplacement()


private void helperHalfs(int i, int j, int offset,
int M)
{
  double s = ((i != 0 ? square[i - offset][j] : 0)
          + (j != 0 ? square[i][j - offset] : 0)
          + (i != M ? square[i + offset][j] : 0)
          + (j != M ? square[i][j + offset] : 0));
  square[i][j] = s / ((i == 0 || j == 0 || j == M
          || i == M) ? 3 : 4) +
  randomDisplacement();
}//end of helperHalfs()


private void calcHalfs()
{
  int offset = (int) ((square.length - 1) /
      Math.pow(2, iter));
  int M = square.length - 1;
  for (int i = 0; i < square.length; i += 2 *
  offset) {
    for (int j = 0; j < square.length; j += 2 *
    offset) {
      i += offset;
      if (i < square.length) {
        helperHalfs(i, j, offset, M);
      }
      i -= offset;
      j += offset;
      if (j < square.length) {
          helperHalfs(i, j, offset, M);
        }
        j -= offset;
        if (sleep != 0) {
          try {
            Thread.sleep(sleep);
          } catch (InterruptedException ex) {
          }
        }
      }
    }
    if (offset != 1) {
      calcMids();
    }
  }//end of calcHalfs()


  private boolean loadTerrain(String path, int dist)
  {
    /*load the generated coordinate file to SketchUp
      and start reddering a terrain model
    */
    lib.AU3_Send("terrain.readFile(\"" +path+ "\", "
        + dist + "){ENTER}", 0);
    lib.AU3_Sleep(3000);
    skp.getDisplay();
    lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
        "");
    // clear the display
    lib.AU3_Sleep(1000);
    lib.AU3_WinActivate(EDITOR_TITLE, "");
    return true;
  }//end of loadTerrain()


  public void setLevel(int level)
  {
    //set the number of texture level, not including
    //the transition levels
    this.level = level;
    if (level <= 0) {
      try {
        throw new TerrainException("Texture level
            must be greater than zero");
      } catch (TerrainException ex) {
        System.err.println("TerrainException: " +
            ex.getMessage());
      }
    }

    int hidLevel = level - 1;
    level += hidLevel;
    tex = new String[level];
    height = new int[level];
    lib.AU3_WinActivate(SKP_TITLE, "");
    lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");

    lib.AU3_Send("terrain.setLevel(" + level +
        "){ENTER}", 0);
    lib.AU3_Sleep(1000);
    skp.getDisplay();
    lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
        "");
    // clear the display

    lib.AU3_Sleep(1000);
    lib.AU3_WinActivate(EDITOR_TITLE, "");
  }//end of setLevel()


  public void setTexture(float num, String texName,
  int height) {
```

```
  //define texture of each texture level of the
  //terrain
  if (num <= 0) {
    try {
      throw new TerrainException("Texture level
         must be greater than zero");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
         ex.getMessage());
    }
  }

  if (num > level) {
    try {
      throw new TerrainException("Texture level is
         out of range");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
         ex.getMessage());
    }
  }
  int index = (int) (num * 2 - 2);
  this.height[index] = height;
  lib.AU3_WinActivate(SKP_TITLE, "");
  lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
  lib.AU3_Send("terrain.setTexture(" + index + ",
         \"" + texName + "\",
         " + height + "){ENTER}", 0);

  skp.getDisplay();
  lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
         "");
  // clear the display
  tex[index] = texName;
  lib.AU3_WinActivate(EDITOR_TITLE, "");
}//end of setTexture()
```

```
public void setTexture(float num, String texName)
throws IOException
{
  //set texture of a flat terrain with no texture
  //levels
  int height = 1;
  if (num <= 0) {
    try {
      throw new TerrainException("Texture level
         must be greater than zero");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
         ex.getMessage());
    }
  }

  if (num > level) {
    try {
      throw new TerrainException("Texture level is
         out of range");
    } catch (TerrainException ex) {
      System.err.println("TerrainException: " +
         ex.getMessage());
    }
  }
  int index = (int) (num * 2 - 2);
  this.height[index] = height;
  lib.AU3_WinActivate(SKP_TITLE, "");
  lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
  lib.AU3_Send("terrain.setTexture(" + index + ",
      \"" + texName + "\",
      " + height + "){ENTER}", 0);

  skp.getDisplay();
  lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
       "");
  // clear the display

  tex[index] = texName;

  lib.AU3_WinActivate(EDITOR_TITLE, "");

}//end of setTexture()


public void texture()
{
  /*create transition level between each texture
    level and sending command to SketchUp to start
    texturing the terrain as defined by method
    setTexture()
  */
  int height;
  for (int i = 1; i < tex.length - 1; i += 2) {
    tex[i] = blendTex(tex[i - 1], tex[i + 1]);
    height = (this.height[i - 1] / 2 +
         this.height[i + 1] / 2) / 2;
    setTexture((float) (i + 2) / 2, tex[i],
         height);

  }
  long startTime = System.currentTimeMillis();
  lib.AU3_WinActivate(SKP_TITLE, "");
  lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");

  lib.AU3_Send("terrain.texture(){ENTER}", 0);
  lib.AU3_Sleep((int) (1000 *
      Math.sqrt(getResolution() / polygonSize)));
  skp.getDisplay();
```

```
  lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
      "");
  // clear the display
  lib.AU3_WinActivate(EDITOR_TITLE, "");
  long stopTime = System.currentTimeMillis();
  System.out.println("Delay = " + (stopTime -
      startTime));

}//end of texture()


private String blendTex(String tex1, String tex2)
{

  //create and write a merged texture file out of
  //two given texture files
  File path = new File("C:/Program Files
      (x86)/SketchUp/SketchUp 2013/Plugins");
  BufferedImage image = ImageIO.read(new
      File(path, tex1));
  BufferedImage overlay = ImageIO.read(new
      File(path, tex2));

  int w = Math.min(image.getWidth(),
      overlay.getWidth());
  int h = Math.min(image.getHeight(),
      overlay.getHeight());
  image = resize(image, w, h);
  overlay = resize(overlay, w, h);
  BufferedImage combined = new BufferedImage(w, h,
      BufferedImage.TYPE_INT_ARGB);

  float alpha = (float) 0.5;
  Graphics2D g = (Graphics2D)
      combined.getGraphics();
  g.drawImage(image, 0, 0, null);
```

```
      g.setComposite(AlphaComposite.getInstance
          (AlphaComposite.SRC_OVER, alpha));
      g.drawImage(overlay, 0, 0, null);

      String name = new String("combined" + texIndex +
           ".png");
      ImageIO.write(combined, "PNG", new File(path,
           name));
      System.out.println("success");
      texIndex++;
      return name;
    }//end of blendTex()


    private static BufferedImage resize(BufferedImage
    img, int newW, int newH)
    {
      //resize a texture image (to be at the same size
      //before merging)
      int w = img.getWidth();
      int h = img.getHeight();
      if (newW == w && newH == h) {
         return img;
      }
      BufferedImage dimg = new BufferedImage(newW,
          newH, img.getType());
      Graphics2D g = dimg.createGraphics();

  g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,

  RenderingHints.VALUE_INTERPOLATION_BILINEAR);
      g.drawImage(img, 0, 0, newW, newH, 0, 0, w, h,
           null);
      g.dispose();
      return dimg;
    }//end of resize()
```

```
public void addModel(Model model)
{
  //add a Model object to a 3D scene on SketchUp
  String[] disp;
  int width, length;
  Point2f position = model.getPosition();
  float z = getHeight(position.x, position.y) +
       model.getOffsetZ();
  model.setZ(z);
  String modelName = "\"" + model.getName() +
      "\"";
  String file = "\"" + model.getFile() + "\"";
  float scale = model.getScale();

  lib.AU3_WinActivate(SKP_TITLE, "");
  lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
  lib.AU3_Send("anim.add_model(" + modelName + ",
      " + file + "," + scale +
      ", " + model.getRotation() + ", " +
      position.x + ", " +
      position.y + ", " + z + ", " +
  model.isMovable() + "){ENTER}", 0);
  lib.AU3_Sleep(1000);
  disp = skp.getDisplay()[1].split(", ");
  width = Integer.parseInt(disp[0].trim());
  length = Integer.parseInt(disp[1].trim());
  model.setWidth(width);
  model.setLength(length);

  lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
      "");
  // clear the display

  lib.AU3_Sleep(1000);
  if (!model.isMovable()) {
    this.models.add(model);
  } else {
    this.movModels.add(model);
  }
  lib.AU3_WinActivate(EDITOR_TITLE, "");
}//end of addModel()


private void saveTerrain(String file, int size)
throws IOException
{
  //write generated coordinates stored in
  //square[][] to a text file
  BufferedWriter out = new BufferedWriter(new
      FileWriter(file));

  out.write("l," + size + "");
  out.newLine();
  for (int i = 0; i < square.length; i++) {
    for (int j = 0; j < square.length; j++) {
      out.write(i + "," + j + "," + (int)
          square[i][j]);
      out.newLine();
    }
  }
  out.close();
}//end of saveTerrain()


private void createPolygon()
{
  //create an array of polygons in the terrain for
  //Barycentric approach
  int i = 0;
  int j = 0;
  int dist = 30;
  float x, y, z;
  Point3f p0, p1, p2, p3;
  QuadArray polygon, polygon2;
  for (i = 0; i < square.length - 1; i++) {

    for (j = 0; j < square.length - 1; j++) {

      polygon = new QuadArray(4,
           QuadArray.COORDINATES);
      polygon2 = new QuadArray(4,
           QuadArray.COORDINATES);
      x = i;
      y = j;
      z = (float) square[i][j];

      p0 = new Point3f(x * dist, y * dist, z);
      p1 = new Point3f(x * dist + dist, y * dist,
          square[i + 1][j]);
      p2 = new Point3f(x * dist, y * dist + dist,
          square[i][j + 1]);
      p3 = new Point3f(x * dist+dist, y * dist +
          dist, square[i+1][j+1]);

      polygon.setCoordinate(0, p0);
      polygon.setCoordinate(1, p2);
      polygon.setCoordinate(2, p3);

      polygon2.setCoordinate(0, p0);
      polygon2.setCoordinate(1, p1);
      polygon2.setCoordinate(2, p3);

      polygonAr.add(polygon);
      polygonAr.add(polygon2);

    }
  }
}//end of createPolygon()


public float getHeight(float x, float y)
{
  /*find height of a particular position on the
    terrain by finding which polygon the
    coordinate is on and then use Bary centric
    approach to calculate the height
  */
  Point3f c0;
  Point3f c1;
  Point3f c2;
  float height = 0;
  int i = 0;
  while (i < polygonAr.size()) {
    c0 = new Point3f();
    c1 = new Point3f();
    c2 = new Point3f();

    polygonAr.get(i).getCoordinate(0, c0);
    polygonAr.get(i).getCoordinate(1, c1);
    polygonAr.get(i).getCoordinate(2, c2);

    height = calcZ(c0, c1, c2, x, y);
    if (height > -1) {
      return height;
    }
    i++;
  }
  return -1;
}//end of getHeight()


public float calcZ(Point3f v1, Point3f v2, Point3f
v3, float x, float y)
  {
    /*use Barycentric theory to calculate height of
      a given coordinate on a
      specific polygon*/
    float det = (v1.y - v3.y)*(v2.x - v3.x) + (v2.y
        - v3.y) * (v3.x - v1.x);
    float b1 = ((y - v3.y)*(v2.x - v3.x) + (v2.y -
        v3.y) * (v3.x - x)) / det;
    float b2 = ((y - v1.y)*(v3.x - v1.x) + (v3.y -
        v1.y) * (v1.x - x)) / det;
    float b3 = 0;

    if ((b1 >= 0) && (b2 >= 0) && (b1 + b2 <= 1)) {
      b3 = 1.0f - b1 - b2;
      return (b1 * v1.z) + (b2 * v2.z) + (b3 *
        v3.z);
    } else {
      return -1;
    }
  }//end of calcZ()


  public void addShape(Shape shape)
  {
    /*add Shape object to the scene including block
      and cylinder; a different command is sent to
      SketchUp accordingly to type of the shape
    */
    Point2f position = shape.getPosition();
    float z = getHeight(position.x, position.y) +
        shape.getOffsetZ();
    shape.setZ(z);
    lib.AU3_WinActivate(SKP_TITLE, "");
    lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
  if (shape.getType().equals("block")) {

    int width = shape.getWidth();
    int length = shape.getLength();
    int height = shape.getHeight();
    lib.AU3_Send("shape.createBlock(\"" +
        shape.getName() + "\", " +
        position.x + ", " + position.y + ", " + z
        + ", " + width + "," +
        length + "," + height + "){ENTER}", 0);
  } else {
    int height = shape.getHeight() +
        shape.getOffsetZ();
    int radius = shape.getRadius();
    lib.AU3_Send("shape.createCylinder(\"" +
        shape.getName() + "\", " +
        position.x + ", " + position.y + ", " + z
        + ", " + radius + ", " +
        height + "){ENTER}", 0);
  }
  this.shape.add(shape);
  lib.AU3_Sleep(1000);
  skp.getDisplay();
  lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
        "");
  // clear the display

  lib.AU3_Sleep(1000);
  lib.AU3_WinActivate(EDITOR_TITLE, "");
}//end of addShape()


public ArrayList<Shape> getShapes()
{return shape;}//end of getShapes()
```

```
public ArrayList<Model> getStaticModels()
{return models;}//end of getModels()


public ArrayList<Model> getDynamicModels()
{return movModels;}//end of getDynamicModels()


public double getResolution()
{return resolution;}// end of getResolution()


public void startAnimation()
{
  //always start animations from time = 0
  startAnimationAt(0);
}//end of startAnimation()


public void startAnimationAt(int time)
{
  //start animations at specific time
  for (int i = 0; i < movModels.size(); i++) {
    movModels.get(i).loadAnimation();
  }
  lib.AU3_WinActivate(SKP_TITLE, "");
  lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
  lib.AU3_Send("anim.animateModels(" + time +
    "){ENTER}", 0);

}//end of startAnimationAt()


public Step[] genStepsArray(Point2f start, Point2f
dest, double sAngle, double dAngle, int startTime,
int stopTime)
```

```
{

  /*generate movement steps array from start,
    destination, start angle, destination angle,
    start time, and stop time of the animation*/
  try {
    return new Movements().genStepsArray(this,
         start, dest, sAngle, dAngle,
         startTime, stopTime);
  } catch (IOException ex) {

    Logger.getLogger(Terrain.class.getName())
    .log(Level.SEVERE, null, ex);
  } catch (MovementException ex) {
    Logger.getLogger(Terrain.class.getName())
    .log(Level.SEVERE, null, ex);
  }
  return null;
}//end of genStepsArray()


public Step[] genStepsArray(Point2f start, Point2f
dest, int startTime, int stopTime)
{
  /*generate steps array only from start,
    destination, start time and stop
    time*/
  try {

    return new Movements().genStepsArray(this,
          start, dest, 0, 0,
          startTime, stopTime);
  } catch (IOException ex) {
    Logger.getLogger(Terrain.class.getName()).
    log(Level.SEVERE, null, ex);
  } catch (MovementException ex) {
    Logger.getLogger(Terrain.class.getName())
    .log(Level.SEVERE, null, ex);
  }
  return null;

}//end of genStepsArray()


public Step[] genStepsArray(Point2f start, Point2f
dest, double dAngle, int startTime, int stopTime)
{
  //generate steps array without a start angle
  try {

    return new Movements().genStepsArray(this,
             start, dest, 0, dAngle,
             startTime, stopTime);
  } catch (IOException ex) {
    Logger.getLogger(Terrain.class.getName()).
    log(Level.SEVERE, null, ex);
  } catch (MovementException ex) {
    Logger.getLogger(Terrain.class.getName()).
    log(Level.SEVERE, null, ex);
  }
  return null;
}//end of genStepsArray()


public Model getModel(String name)
{ //get model by name
  for (Model model : movModels) {
    if (model.getName().equals(name)) {
      return model;
    }
  }
  return null;
  }//end of getModel()


  public void enableShadow()
  {
    lib.AU3_Send("Sketchup.send_action
10602{ENTER}", 0);
  }// end of enableShadow()


  public void loadStatics(String fnm)
  /* The format of the input lines are:
   b block-name x y width length height rotation
   texture-fnm [roof-texture]
   c cylinder-name x y radius height rotation
   texture-fnm
   m model-fnm model-name x y scale rotation
   and blank lines and comment lines.
   */ {
    System.out.println("Reading file: " + fnm);
    try {
      BufferedReader br = new BufferedReader(new
          FileReader(fnm));
      String line;
      String[] toks;
      StaticShape ss = null;
      int shapeLineCount = 1;
      // only counts lines with shape data on them
      while ((line = br.readLine()) != null) {
        if (line.length() == 0) // blank line
        {
          continue;
        }
        if (line.startsWith("//")) // comment
        {
          continue;
        }

        // System.out.println("Line: " + line);
        toks = line.split("\\s+");
        if (toks == null) {
          System.out.println("Could not tokenize
                shape " + shapeLineCount +
                ": \"" + line + "\"");
          shapeLineCount++;
          continue;
        }

        char staticCh =
            toks[0].toLowerCase().charAt(0);
        ss = null;
        if (staticCh == 'b') {
          createBlock(toks, shapeLineCount);
        } else if (staticCh == 'c') {
          createCylinder(toks, shapeLineCount);
        } else if (staticCh == 'm') {

          createModel(toks, shapeLineCount);

        } else {
          System.out.println("Did not recognize
              static type for shape " +
              shapeLineCount + ": \"" + line +
              "\"");
        }
        shapeLineCount++;
        if (ss != null) {
          statics.add(ss);
        }
      }
      br.close();
    System.out.println("Number of static shapes
         created: " + statics.size());
  } catch (IOException e) {
    System.out.println("Error reading file: " +
         fnm);
  }
}  // end of loadInfo()


private void createBlock(String[] toks, int slc)
 /* Extract data for toks of the form:
 b block-name x y width length height rotation
 texture-fnm [roof-texture]
 */ {
  if ((toks.length < 9) || (toks.length > 10)) {
    System.out.println("Wrong number of block
    toks; skipping shape " +
    slc);
  }

  String roofFnm = ((toks.length == 10) ? toks[9]
      : null);
  StaticBlock blockInfo = new StaticBlock(toks[1],
          getD(toks[2]), getD(toks[3]),
          getD(toks[4]), getD(toks[5]),
          getD(toks[6]),
          getD(toks[7]), toks[8], roofFnm);
  Block block = new Block(new
          Point2f(blockInfo.getX(),
          blockInfo.getY()),
        (int) blockInfo.getWidth(), (int)
         blockInfo.getLength(),
        (int) blockInfo.getHeight());
  addShape(block);
  if (toks.length == 10) {
    block.texture(blockInfo.getTextureFnm(),
```

```
      blockInfo.getRoofFnm());
    } else {
      block.texture(blockInfo.getTextureFnm());
    }
  }  // end of getBlockInfo()


  private void createCylinder(String[] toks, int
  slc)
   /* Extract data for toks of the form:
      c cylinder-name x y radius height rotation
      texture-fnm
   */ {
    if (toks.length != 8) {
      System.out.println("Wrong number of cylinder
      toks; skipping shape " +
      slc);

    } else {
      StaticCylinder cylinderInfo = new
              StaticCylinder(toks[1],
              getD(toks[2]), getD(toks[3]),
              getD(toks[4]), getD(toks[5]),
              getD(toks[6]), toks[7]);
      Cylinder cylinder = new Cylinder(new
              Point2f(cylinderInfo.getX(),
              cylinderInfo.getY()), (int)
              cylinderInfo.getRadius(), (int)
              cylinderInfo.getHeight());
      addShape(cylinder);

cylinder.texture(cylinderInfo.getTextureFnm());
    }
  }// end of getCylinderInfo()
  private void createModel(String[] toks, int slc)
   /* Extract data for toks of the form:
   m model-fnm model-name x y scale rotation
   */ {
    if (toks.length != 7) {
      System.out.println("Wrong number of model
      toks; skipping shape " +
                      slc);
    } else {

      StaticModel modelInfo = new
      StaticModel(toks[2], getD(toks[3]),
              getD(toks[4]),
              getD(toks[5]), getD(toks[6]),
              toks[1]);

      Model model = new
          Model(modelInfo.getModelFnm(),
          new Point2f(modelInfo.getX(),
          modelInfo.getY()), (float)
          modelInfo.getScale(),
          modelInfo.getRotation(), false);
      addModel(model);
    }
  }  // end of getModelInfo()


  private float getD(String token)
  {
    float num = 0;
    try {
      num = Float.parseFloat(token);
    } catch (NumberFormatException ex) {
      System.out.println("Incorrect format for " +
token);
    }
    return num;
  }  // end of getD()


  public ArrayList<QuadArray> getPolygonArray()
  {return polygonAr;}

}//end of Terrain
```

Appendix B.5 User.Java

```
//contains methods for creating user model and
//setting camera to the model
public class User extends Model {

  private static SKPControl skp = new SKPControl();
  private static final String SKP_TITLE = "Untitled
     - SketchUp";
  private static final String EDITOR_TITLE =
     "Output-terrain2(run)- Editor";
  private static final String RC_TITLE = "Ruby
      Console";
  private static JAutoIt lib = JAutoIt.INSTANCE;

  public boolean setCam(Point3d eye)
  {
    //set the camera to follow the user model
    Point2f position = getPosition();
    float height = getHeight();
    lib.AU3_WinActivate(SKP_TITLE, "");
    lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
    lib.AU3_Send("anim.setCam(" + eye.x + "," +
        eye.y + "," + eye.z + "," +
        position.x + "," + position.y + "," + height
        + ")", 1);
    lib.AU3_Send("{ENTER}", 0);
    lib.AU3_Send("anim.setDynamicCam(true)", 1);
    lib.AU3_Send("{ENTER}", 0);
    lib.AU3_Sleep(1000);
    skp.getDisplay();
    lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
       "");
    // clear the display

    lib.AU3_Sleep(1000);
    lib.AU3_WinActivate(EDITOR_TITLE, "");
    return true;
  }//end of setCam()

  public User(String file, Point2f position, int
  offsetZ, float scale, float rotation)
  {super(file, position, offsetZ, scale, rotation,
   true);}

}//end of User class
```

Appendix B.6 Shape.Java

```
/*Abstract class for cylinder and block containing
methods for getting and   setting parameters and
also texturing methods for shapes*/

public class Shape extends Model {

  private int radius;
  private String type;
  private JAutoIt lib = JAutoIt.INSTANCE;
  private SKPControl skp = new SKPControl();
  private String SKP_TITLE = "Untitled - SketchUp";
  private String EDITOR_TITLE = "Output - terrain2
     (run)   - Editor";
  private String RC_TITLE = "Ruby Console";
  private int offsetZ = 0;

  public void setRadius(int radius)
  {this.radius = radius;}


  public int getRadius()
  {return radius;}


  public void setType(String type)
  {this.type = type;}


  public String getType()
  {return type;}


  public Shape()
  {super("", new Point2f(0, 0), 0, 1, 0, false);}


  public boolean texture(String sideTex)
  { //apply sideTex to the side of shape and black
    //color to the roof
    String roof = "black.jpg";
    lib.AU3_WinActivate(SKP_TITLE, "");
    lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
    lib.AU3_Send("shape.texture(\"" + getName() +
       "\", \"" + roof + "\", \""
       + sideTex + "\") {ENTER}", 0);
    skp.getDisplay();
    lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
       "");
    // clear the display
    lib.AU3_Sleep(1000);
    lib.AU3_WinActivate(EDITOR_TITLE, "");
    return true;
  }//end of texture()


  public boolean texture(String roof, String side)
  {
    // apply side texture and roof texture to a
    // shape
    lib.AU3_WinActivate(SKP_TITLE, "");
    lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");
    lib.AU3_Send("shape.texture(\"" + getName() +
        "\", \"" + roof + "\", \""
        + side + "\") {ENTER}", 0);
    skp.getDisplay();
    lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
        "");
    // clear the display
    lib.AU3_Sleep(1000);
    lib.AU3_WinActivate(EDITOR_TITLE, "");
    return true;
  }//end of texture()

}//end of Shape class
```

Appendix B.7 Block.Java

```
/**provide two block constructor for generating a
   Block instance with and without specifying Z
   offset of the block
 */
public class Block extends Shape {

  private static int count = 0;

  public Block(Point2f position, int offsetZ, int
  width, int length, int height)
  {
    //create block instance with position, offsetZ,
    //and block dimension
    if (width <= 0) {
      try {
        throw new ShapeException("Block width cannot
           be negative or zero");
      } catch (ShapeException ex) {
        System.err.println("ShapeException: " +
           ex.getMessage());
      }
    }
    if (length <= 0) {
      try {
        throw new ShapeException("Block length
           cannot be negative or zero");
      } catch (ShapeException ex) {
        System.err.println("ShapeException: " +
           ex.getMessage());
      }
    }
    if (height <= 0) {
      try {
```

```
      throw new ShapeException("Block height
         cannot be negative or zero");
    } catch (ShapeException ex) {
      System.err.println("ShapeException: " +
         ex.getMessage());
    }
  }
  String name = "Block" + count;
  setName(name);
  setPosition(position);
  setWidth(width);
  setLength(length);
  setHeight(height);
  setType("block");
  setOffsetZ(offsetZ);
  count++;
}//end of Block constructor


public Block(Point2f position, int width, int
length, int height)
{
  //create block instance without offsetZ
  if (width <= 0) {
    try {
      throw new ShapeException("Block width cannot
          be negative or zero");
    } catch (ShapeException ex) {
      System.err.println("ShapeException: " +
          ex.getMessage());
    }
  }
  if (length <= 0) {
    try {
      throw new ShapeException("Block length
          cannot be negative or zero");
    } catch (ShapeException ex) {
      System.err.println("ShapeException: " +
          ex.getMessage());
    }
  }
  if (height <= 0) {
    try {
      throw new ShapeException("Block height
         cannot be negative or zero");
    } catch (ShapeException ex) {
      System.err.println("ShapeException: " +
         ex.getMessage());
    }
  }
  String name = "Block" + count;
  setName(name);
  setPosition(position);
  setWidth(width);
  setLength(length);
  setHeight(height);
  setType("block");
  setOffsetZ(0);
  count++;
}//end of Block constructor

}//end of Block
```

Appendix B.8 Cylinder.Java

```
/**
 * provide constructors for creating Cylinder
 *
 */
public class Cylinder extends Shape {
```

```
private static int count = 0;

public Cylinder(Point2f position, int offsetZ, int
radius, int height)
{ //create cylinder instance from position, Z
  //offset, radius, and height
  if (radius <= 0) {
    try {
      throw new ShapeException("Cylinder radius
         cannot be negative or zero");
    } catch (ShapeException ex) {
      System.err.println("ShapeException: " +
         ex.getMessage());
    }
  }
  if (height <= 0) {
    try {
      throw new ShapeException("Cylinder height
         cannot be negative or zero");
    } catch (ShapeException ex) {
      System.err.println("ShapeException: " +
         ex.getMessage());
    }
  }
  String name = "Cylinder" + count;
  setName(name);
  setPosition(position);
  setRadius(radius);
  setHeight(height);
  setType("cylinder");
  setOffsetZ(offsetZ);
  count++;
}//end of Cylinder constructor


public Cylinder(Point2f position, int radius, int
  height)
  { //create cylinder without Z offset
    if (radius <= 0) {
      try {
        throw new ShapeException("Cylinder radius
           cannot be negative or zero");
      } catch (ShapeException ex) {
        System.err.println("ShapeException: " +
           ex.getMessage());
      }
    }
    if (height <= 0) {
      try {
        throw new ShapeException("Cylinder height
            cannot be negative or zero");
      } catch (ShapeException ex) {
        System.err.println("ShapeException: " +
            ex.getMessage());
      }
    }
    String name = "Cylinder" + count;
    setName(name);
    setPosition(position);
    setRadius(radius);
    setHeight(height);
    setType("cylinder");
    setOffsetZ(0);
    count++;

  }//end of Cylinder constructor

}//end of Cylinder
```

Appendix B.9 Movements.Java

```
/**
 *
 * generate steps array for dynamic models
 */
public class Movements {

  private ArrayList<QuadArray> polygonAr;
  private ArrayList<Point3f> steps = new
      ArrayList<Point3f>(); //no height
  private ArrayList<Step> steps2 = new
      ArrayList<Step>(); //has height
  private float subAngle;
  private int totalStep;
  private Step[] stepsArray;
  private Terrain terrain;


  public Step[] genStepsArray(Terrain terrain,
  Point2f start, Point2f dest, double sAngle, double
  dAngle, int startTime, int stopTime)
  {
    /*calculate position and rotation of each
      coordimate along the animation
      path from start tp dest
    */
    if (startTime < 0 || stopTime < 0) {
      try {
        throw new MovementException("Time cannot be
           negative number");
      } catch (MovementException ex) {
        System.err.println("MovementException: "+
           ex.getMessage());
      }
     }
if (startTime > stopTime) {
  try {
    throw new MovementException("Stop time must
        be greater than start time");
  } catch (MovementException ex) {
    System.err.println("MovementException: "+
        ex.getMessage());
  }
}
this.terrain = terrain;
polygonAr = terrain.getPolygonArray();
int step = stopTime - startTime;
totalStep = step;
subAngle = (float) (dAngle - sAngle) / (step);
int index = 0;
Point3f t1;
float h;
Point3f prevP = new Point3f();
int preIndex = 0;
//calculate x, y position of each coordinate
calcStep(start, dest);
prevP = steps.get(0);
//find the intersection coodinate and store
//every coordinate in steps2
for (int i = 0; i < steps.size(); i++) {
  t1 = new Point3f(steps.get(i).x,
      steps.get(i).y, 0);
  h = terrain.getHeight((int) t1.x, (int) t1.y);
  index = findPolygon((int) t1.x, (int) t1.y);
  steps.get(i).setZ(h);
  //if different polygon
  if (i > 0 && index != preIndex) {
    //calculate intersection and add to steps2
    calcIntersection(prevP, t1, index,
       startTime);
    }
    steps2.add(new Step(startTime, steps.get(i).x,
       steps.get(i).y, h, 0));
    startTime++;
    prevP = steps.get(i);
    preIndex = index;
  }
  //crate steps array from steps2
  stepsArray = new Step[steps2.size()];
  for (int i = 0; i < steps2.size(); i++) {
    stepsArray[i] = steps2.get(i);
  }
  //calculate angle at each coordinate of array
  calAngle();
  System.out.println("Movements are generated");
  return stepsArray;

}//end of genStepsArray


private int calcStep(Point2f start, Point2f end)
{
  /*calculate small vector from start position to
    next coordinate and increment the start
    position by the vector in each iteration of
    the loop, in order to calculate each
    coordinate along the path to the destination
  */
  float d = 0;
  float px = 0, py = 0;
  int count = 0;
  float vx = end.x - start.x;
  float vy = end.y - start.y;
  d = (float) Math.sqrt(Math.pow(vx, 2) +
       Math.pow(vy, 2));
```

```
float step = d / (totalStep);
/*calculate each coordinate from the vector and
  check if the coordinate is in the terrain and
  if there is a static collision, before adding
  the coordinate to the array
  **static collision will raise and exception**
*/
for (int i = 0; i <= totalStep; i++) {
  if (vx == 0 && vy == 0) {
    steps.add(new Point3f(start.x, start.y, 0));

  } else {
    if (i == 0) {
      if (isCollision(start.x, start.y) &&
         !isOutOfTerrain(start.x, start.y))
      {
        steps.add(new Point3f(start.x, start.y,
          0));
      } else {
        break;
      }

    } else if (i == totalStep) {
      if (!isCollision(end.x, end.y) &&
          !isOutOfTerrain(end.x, end.y)) {
        steps.add(new Point3f(end.x, end.y, 0));
      } else {
        break;
      }

    } else {
      step *= i;
      px = (float) (start.x + vx * (step / d));
      py = (float) (start.y + vy * (step / d));
      step /= i;
```

```
        if (!isCollision(px, py) &&
            !isOutOfTerrain(px, py)) {
          steps.add(new Point3f(px, py, 0));

        } else {
          break;
        }
      }
    }
    count++;
  }
  return count;

}//end of calcStep


private void calcIntersection(Point3f p0, Point3f
p1, int i, int time)
{
  /*
  get three sides of polygon where p0 is on and
  try calculating intersection between vector p0p1
  and each side of the polygon
  */
  Point3f c0 = new Point3f(),
          c1 = new Point3f(),
          c2 = new Point3f();

  polygonAr.get(i).getCoordinate(0, c0);
  polygonAr.get(i).getCoordinate(1, c1);
  polygonAr.get(i).getCoordinate(2, c2);

  lineIntersect(c0.x, c0.y, c1.x, c1.y, p0.x,
      p0.y, p1.x, p1.y, time);
  lineIntersect(c0.x, c0.y, c2.x, c2.y, p0.x,
      p0.y, p1.x, p1.y, time);
  lineIntersect(c1.x, c1.y, c2.x, c2.y, p0.x,
      p0.y, p1.x, p1.y, time);
}//end of calcIntersection


private int lineIntersect(float x1, float y1,
    float x2, float y2, float x3,
    float y3, float x4, float y4, int time)
{
 /*calculate intersection between four coordinates
   representing two vectors*/
  double denom = (y4 - y3) * (x2 - x1) - (x4 - x3)
      * (y2 - y1);
  Point3f p;
  float h;
  if (denom == 0.0) { //lines are parallel.
    return 0;
  }
  double ua = ((x4 - x3) * (y1 - y3) - (y4 - y3) *
      (x1 - x3)) / denom;
  double ub = ((x2 - x1) * (y1 - y3) - (y2 - y1) *
      (x1 - x3)) / denom;
  if (ua >= 0.0f && ua <= 1.0f && ub >= 0.0f && ub
     <= 1.0f) {
    //get the intersection point.
    p = new Point3f((float) (x1 + ua * (x2 - x1)),
          (float) (y1 + ua * (y2 - y1)), 0);
    h = terrain.getHeight((float) p.x, (float)
           p.y);
    p.setZ(h);
    steps2.add(new Step((float) (time - 0.5), p.x,
           p.y, p.z, 0));
    return 1;

  }
  return 0;
```

```
}//end of line Intersect


private void calAngle()
{
  //assign angle to each point in the steps array
  float angle = 0;
  for (int i = 0; i < stepsArray.length; i++) {
    if (i == 0 || i == steps.size()) {
      angle = 0;
    } else {
      angle = subAngle;
    }
    stepsArray[i].setAngle(angle);
  }
}//end of calAngle()


private boolean isOutOfTerrain(float x, float y)
{
  //chech whether the coordinate is out of the
  //terrain
  double edge = terrain.getResolution();
  if (x > edge || y > edge) {
    try {
      throw new MovementException("Edge is
           detected at " + (int) x + "," +
          (int) y);
    } catch (MovementException ex) {
      System.err.println("MovementException: "+
           ex.getMessage());
    }
  }
  return false;
}//end of isOutOfTerrain
```

```
private boolean isCollision(float x, float y)
{
  //find if there is a static collision at the
  //given coordinate
  ArrayList<Shape> shapes;
  ArrayList<Model> models;
  int i, width, length;
  Point2f position;
  //collision with shapes
  if ((shapes = terrain.getShapes()) != null) {
    for (i = 0; i < shapes.size(); i++) {

      width = shapes.get(i).getWidth() / 2;
      length = shapes.get(i).getLength() / 2;
      position = shapes.get(i).getPosition();

      if (x >= position.x - width && x <=
        position.x + width) {
        if (y >= position.y - length && y <=
        position.y + length) {
          try {
            throw new MovementException("Collision
                  is detected at " + x +
                  ", " + y);
          } catch (MovementException ex) {
            System.err.println("MovementException:
                  "+ ex.getMessage());
          }

        }
      }
    }
  }
  //collision with static models
```

```
    if ((models = terrain.getStaticModels()) !=
    null) {
      Point3f c0, c1, c2, c3;

      for (i = 0; i < models.size(); i++) {

        width = models.get(i).getWidth() / 2;
        length = models.get(i).getLength() / 2;
        position = models.get(i).getPosition();
        c0 = new Point3f(position.x - width,
             position.y - length, 0);
        c1 = new Point3f(position.x - width,
             position.y + length, 0);
        c2 = new Point3f(position.x + width,
             position.y + length, 0);
        c3 = new Point3f(position.x + width,
             position.y - length, 0);

        if ((terrain.calcZ(c0, c1, c2, x, y) != -1)
        || (terrain.calcZ(c0, c2, c3, x, y) != -1))
        {
          try {
            throw new MovementException("Collision
                  is detected at " + x + ",
                  " + y);
          } catch (MovementException ex) {
            System.err.println("MovementException:
                 "+ ex.getMessage());
          }
        }
      }
    }
    return false;
  }//end of isCollision()
```

```
  public int findPolygon(float x, float y)
  {
    //find which polygon the coordinate is on using
    //Bary centric approach
    Point3f c0;
    Point3f c1;
    Point3f c2;

    int i = 0;
    while (i < polygonAr.size()) {
      c0 = new Point3f();
      c1 = new Point3f();
      c2 = new Point3f();

      polygonAr.get(i).getCoordinate(0, c0);
      polygonAr.get(i).getCoordinate(1, c1);
      polygonAr.get(i).getCoordinate(2, c2);

      if (terrain.calcZ(c0, c1, c2, x, y) > -1) {
        return i;
      }
      i++;
    }
    return -1;
  }//end of findPolygon()

}//end of Movement class
```

Appendix B.10 Step.Java

```
/**
 * store information in each step of the model's
 * animation including time, x,
 * y, z, and angle
 */
```

```
public class Step {

  private float time;
  private float x;
  private float y;
  private float z;
  private float angle;

  public float getX()
  {return x;}


  public float getY()
  {return y;}


  public float getZ()
  {return z;}


  public float getAngle()
  {return angle;}


  public float getTime()
  {return time;}


  public void setX(float x)
  {this.x = x;}


  public void setY(float y)
  {this.y = y;}


  public void setZ(float z)
  {this.z = z;}


  public void setAngle(float angle)
  {this.angle = angle;}


  public void setTime(int time)
  {this.time = time;}


  public Step(float time, float x, float y, float z,
  float angle)
  {
    this.time = time;
    this.x = x;
    this.y = y;
    this.z = z;
    this.angle = angle;
  }
}//end of Step
```

Appendix B.11 CollisionEvent.Java

```
/*
  Read collision file and and store collision
  information. The models names are utilized to open
  their steps file, regenerate steps arrays of
  source and target and store in the CollisionEvent
  instance.
 */
public class CollisionEvent {

  private String name, targetName;
private int x0, y0, w0, h0, x1, y1, w1, h1, vx,
   vy, time;
float angle, tAngle;
private ArrayList<String[]> sourceFile = new
   ArrayList<String[]>();
private ArrayList<String[]> targetFile = new
   ArrayList<String[]>();
private Step[] sourceSteps, targetSteps;
private static SKPControl skp = new SKPControl();
private String path;

public CollisionEvent(Object source, String file)
{ /*read collision file and create steps array out
    of each model's step file*/
  path = skp.getDirectory();
  BufferedReader br = null;
  try {
    br = new BufferedReader(new FileReader(path +
         file));
  } catch (FileNotFoundException ex) {
    System.err.println("FileNotFoundException: "+
         ex.getMessage());
  }
  String line;
  String[] collisionInfo = null;
  //extract collision information
  try {
    while ((line = br.readLine()) != null) {
      collisionInfo = line.split(", ");
      angle = Float.parseFloat(collisionInfo[13]);
      tAngle =
          Float.parseFloat(collisionInfo[14]);
      time = Integer.parseInt(collisionInfo[12]);
      w0 = Integer.parseInt(collisionInfo[1]);
      h0 = Integer.parseInt(collisionInfo[2]);
      x0 = Integer.parseInt(collisionInfo[3]);
      y0 = Integer.parseInt(collisionInfo[4]);
      w1 = Integer.parseInt(collisionInfo[6]);
      h1 = Integer.parseInt(collisionInfo[7]);
      x1 = Integer.parseInt(collisionInfo[8]);
      y1 = Integer.parseInt(collisionInfo[9]);

      name = collisionInfo[0];
      targetName = collisionInfo[5];

    }
  } catch (IOException ex) {
    System.err.println(“IOException: ”+
        ex.getMessage());
  }

  //read step files
  String[] stepInfo;
  try {
    BufferedReader step = new BufferedReader(new
          FileReader(path + name + "Step.txt"));
    BufferedReader tarStep = new
    BufferedReader(new FileReader(path +
                 targetName + "Step.txt"));
    while ((line = step.readLine()) != null) {
      stepInfo = line.split(", ");
      sourceFile.add(stepInfo);
    }
    while ((line = tarStep.readLine()) != null) {
      stepInfo = line.split(", ");
      targetFile.add(stepInfo);
    }
    step.close();
    tarStep.close();
    //reset the files
    new FileWriter(path + name + "Step.txt");
    new FileWriter(path + targetName +
        "Step.txt");

  } catch (IOException ex) {
    System.err.println(“IOException: ”+
        ex.getMessage());
  }
  sourceSteps = new Step[sourceFile.size()];
  targetSteps = new Step[targetFile.size()];
  int time;
  float x, y, z, angle;
  //recreate source's steps array
  for (int i = 0; i < sourceFile.size(); i++) {
    time = Integer.parseInt(sourceFile.get(i)[0]);
    x = Integer.parseInt(sourceFile.get(i)[1]);
    y = Integer.parseInt(sourceFile.get(i)[2]);
    z = Integer.parseInt(sourceFile.get(i)[3]);
    angle =
       Integer.parseInt(sourceFile.get(i)[4]);
    sourceSteps[i] = new Step(time, x, y, z,
       angle);
  }
  //create target's steps array
  for (int i = 0; i < targetFile.size(); i++) {
    time = Integer.parseInt(targetFile.get(i)[0]);
    x = Integer.parseInt(targetFile.get(i)[1]);
    y = Integer.parseInt(targetFile.get(i)[2]);
    z = Integer.parseInt(targetFile.get(i)[3]);
    angle =
       Integer.parseInt(targetFile.get(i)[4]);
    targetSteps[i] = new Step(time, x, y, z,
       angle);
  }
}//end of CollisionEvent constructor
```

```
public String getSourceName()
{ return name; }

public String getTargetName()
{ return targetName; }

public Point2f getSourcePos()
{ return new Point2f(x0, y0); }

public Point2f getTargetPos()
{ return new Point2f(x1, y1); }

public Point2f getSourceDimen()
{ return new Point2f(w0, h0); }

public Point2f getTargetDimen()
{ return new Point2f(w1, h1); }

public float getTargetAngle()
{ return tAngle; }

public int getTime()
{ return time; }

public float getSourceAngle()
{ return angle; }
```

```
  public Step[] getSourceStep()
  { return sourceSteps; }


  public Step[] getTargetStep()
  { return targetSteps; }


  public Step getSourceStepAt(int time)
  { return sourceSteps[time]; }


  public Step getTargetStepAt(int time)
  { return targetSteps[time]; }

}//end of CollisionEvent
```

Appendix B.12 CollisionRecovery.Java

```
/**
 * solve a collision by updating steps array of
   source and target in order to have target waits
   for small amount of time, while source is
   reversing and moving around the target. The class
   contains methods for recovering a collision and
   manipulating steps array of the model
 */
public class CollisionRecovery {

  Step[] stepsArray;
  static Step[] modelStep;
  TerrainConstructor terrain;
  private static SKPControl skp = new SKPControl();
  private static String EDITOR_TITLE = "Output -
     terrain2 (run)   - Editor";
private static String RC_TITLE = "Ruby Console";
private static JAutoIt lib = JAutoIt.INSTANCE;

public static void
     basicRecovery(TerrainConstructor terrain,
CollisionEvent e)
{
  /* use collision information to calculate
     coordinate P1, P2, P3, and P4.
     Then update steps array of source and target
     in order to make source model moves to P1-
     P4(around the target), while the target is
     waiting.
  */
  Step[] targetStep;
  float angle = e.getSourceAngle();
  int time = e.getTime();
  //model param
  modelStep = e.getSourceStep();
  Point2f p0 = e.getSourcePos();
  Point2f dimen = e.getSourceDimen();

  //target param
  targetStep = e.getTargetStep();
  Point2f tarPosition = e.getTargetPos();
  Point2f tarDimen = e.getTargetDimen();

  Step[] mov1 = null, mov2 = null, mov3 = null,
     mov4 = null, mov5 = null, mov6 = null, mov7 =
     null, mov8 = null, mov9 = null, mov10 = null;
  Step[] tarMov1 = null, tarMov2 = null;
  Point2f p1 = null, p2 = null, p3 = null, p4 =
     null;

  float offsetX = 0, offsetY = 0, reverse = 0;
```

```
int rot = -90;
int p4Index = 0;
reverse = dimen.x / 2;
offsetX = dimen.x + tarDimen.x + reverse;
offsetY = dimen.y + tarDimen.y;
p1 = new Point2f(
      (p0.x - reverse *
       Math.cos(Math.toRadians(angle))),
      (p0.y - reverse *
       Math.sin(Math.toRadians(angle))));
p2 = new Point2f((
      (p1.x + offsetY *
       Math.cos(Math.toRadians(90 - angle))),
      (p1.y - offsetY *
       Math.sin(Math.toRadians(90 - angle))));
p3 = new Point2f((p2.x + offsetX *
       Math.cos(Math.toRadians(angle))),
      (p2.y + offsetX *
       Math.sin(Math.toRadians(angle))));
p4Index = findClosestPoint(new Point2f(
      (p1.x + offsetX *
       Math.cos(Math.toRadians(angle))),
      (p1.y + offsetX *
       Math.sin(Math.toRadians(angle)))),
       modelStep);
p4 = new Point2f(modelStep[p4Index].getX(),
       modelStep[p4Index].getY());
//update source steps array
int tarIndex = findClosestPoint(new
       Point2f(tarPosition.x,
       tarPosition.y), targetStep);
mov1 = terrain.genStepsArray(p0, p1, 0, 0, time,
       time + 1);
mov2 = terrain.genStepsArray(p1, p1, 0, rot,
       time + 1, time + 2);
mov3 = terrain.genStepsArray(p1, p2, 0, 0, time
       + 2, time + 3);
mov4 = terrain.genStepsArray(p2, p2, 0, -1 *
       rot, time + 3, time + 4);
mov5 = terrain.genStepsArray(p2, p3, 0, 0, time
       + 4, time + 5);
mov6 = terrain.genStepsArray(p3, p3, 0, -1 *
       rot, time + 5, time + 6);
mov7 = terrain.genStepsArray(p3, p4, 0, 0, time
       + 6, time + 7);
mov8 = terrain.genStepsArray(p4, p4, 0, rot,
       time + 7, time + 8);
mov9 = extractArray(p4Index, time + 8,
       modelStep);
//update target steps array
tarMov1 = terrain.genStepsArray(tarPosition,
       tarPosition, 0, 0, time - 1, time + 2);
tarMov2 = extractArray(tarIndex, time + 2,
       targetStep);

//load animation to SketchUp side
for (Model model : terrain.getDynamicModels()) {
  if (model.getName().equals(e.getSourceName()))
{
    model.setAnimation(mov1);
    model.setAnimation(mov2);
    model.setAnimation(mov3);
    model.setAnimation(mov4);
    model.setAnimation(mov5);
    model.setAnimation(mov6);
    model.setAnimation(mov7);
    model.setAnimation(mov8);
    model.setAnimation(mov9);
  }
  if (model.getName().equals(e.getTargetName()))
{
    model.setAnimation(tarMov1);
```

```
        model.setAnimation(tarMov2);
      }
    }
    //resume animations at 'time'
    terrain.startAnimationAt(time);
  }//end of basicRecovery()


  public static Step[] extractArray(int index, int
  time, Step[] stepsArray)
  {
    /*extract subarray from stepsArray started from
      the given index and update the time of each
      element of the array started from 'time'
    */
    int count = 0;
    int size = stepsArray.length - index;
    float x, y, z, angle;
    Step step;
    Step[] newSteps = new Step[size];
    for (int i = index; i < stepsArray.length; i++)
{
      step = stepsArray[i];
      x = step.getX();
      y = step.getY();
      z = step.getZ();
      angle = step.getAngle();
      newSteps[count] = new Step(time + count, x, y,
         z, angle);
      count++;
    }
    return newSteps;
  }//end of extractArray()


  public static int findClosestPoint(Point2f p4,
```

```
Step[] stepsArray)
{
//return the index of position closest to given p4
  Point2f p;
  int index = 0;
  Point2f start = new
      Point2f(stepsArray[0].getX(),
      stepsArray[0].getY());
  float dist = start.distance(p4);

  for (int i = 1; i < stepsArray.length; i++) {
    p = new Point2f(stepsArray[i].getX(),
        stepsArray[i].getY());
    if (p.distance(p4) <= dist) {
      dist = p.distance(p4);
      index = i;
    } else {
      break;
    }
  }
  return index;
}//end of findClosestPoint()


public static boolean stopModel(String name)
{
  //stop the animation of the model
  lib.AU3_Send("anim.stopModel(\"" + name + "\")
     {ENTER}", 0);
  skp.getDisplay();
  lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
     "");
  lib.AU3_Sleep(1000);
  lib.AU3_WinActivate(EDITOR_TITLE, "");
  return true;
}//end of stopModel()
```

```
}//end of CollisionRecovery
```

Appendix B.13 AnimThread.Java

```
/*
  Monitor for creation and modification of a file in
  project directory, create event object based on
  the file name, and call event handling method
  for handling the event
 */
public class AnimThread implements Runnable {

  private Thread runner;
  private boolean start = false;
  private Path path;
  private boolean isAnimationStop = false;
  private boolean isCollision = false;
  private AnimationListener lis;

  public AnimThread(Path path)
  {
    this.path = path;
  }


  public AnimThread(String threadName)
  {
    //create and start new thread
    runner = new Thread(this, threadName);
    System.out.println(runner.getName());
    runner.start();
  }
public void run()
{
  System.out.println("Animation thread is
      started");
  // Obtain the file system of the Path
  FileSystem fs = path.getFileSystem();
  // Create the new WatchService
  try (WatchService service =
      fs.newWatchService()) {

    // Register the path to the service
    // Watch for creation events
    path.register(service, ENTRY_CREATE,
        ENTRY_DELETE, ENTRY_MODIFY);

    // Start the infinite polling loop
    WatchKey key = null;
    while (true) {
      key = service.take();

      // Dequeueing events
      Kind<?> kind = null;
      for (WatchEvent<?> watchEvent :
        key.pollEvents()) {
        // Get the type of the event
        kind = watchEvent.kind();

        if (ENTRY_MODIFY == kind) {
          Path evPath = ((WatchEvent<Path>)
              watchEvent).context();
          // Output
          String fileName =
              evPath.toFile().getName();
          if (fileName.equals("Message.txt")) {
            try {
              String line;
        BufferedReader br = new
           BufferedReader(new
             FileReader(evPath.toString()));
        while ((line = br.readLine()) !=
        null) {
          System.out.println(line);
        }
      } catch (FileNotFoundException ex) {

          System.err.println("
          FileNotFoundException: " +
          ex.getMessage());
      }
    }
    if (fileName.contains("Collision")) {
      isCollision = true;
      lis.collision(new CollisionEvent(this,
          fileName));
    }
  }
  if (ENTRY_CREATE == kind) {
    // A new Path was created
    Path evPath = ((WatchEvent<Path>)
        watchEvent).context();
    // Output
    String fileName =
        evPath.toFile().getName();
    if (fileName.contains("Collision")) {
      isCollision = true;
      lis.collision(new CollisionEvent(this,
          fileName));
    }
    if (fileName.equals("finish.txt")) {
      lis.finish(new FinishEvent(this));
      isAnimationStop = true;
    }
```

```
        }
        if (!key.reset()) {
          break; //loop
        }
      }
    }

  } catch (IOException ex) {
    System.err.println("IOException: " +
        ex.getMessage());
  } catch (InterruptedException ex) {
    System.err.println("Interruptedxception: " +
        ex.getMessage());
  }

}//end of run()


public boolean isFinished()
{return isAnimationStop;}


public boolean getCollisionStat()
{return isCollision;}


public void setListener(AnimationListener
listener)
{lis = listener;}


public boolean isStarted()
{return start;}

}//end of AnimThread class
```

Appendix B.14 Scene.Java

```
public class Scene {

  private static SKPControl skp = new SKPControl();
  private static String SKP_TITLE = "Untitled -
     SketchUp";
  private static String EDITOR_TITLE = "Output -
     terrain2 (run)   - Editor";
  private static String RC_TITLE = "Ruby Console";
  private static JAutoIt lib = JAutoIt.INSTANCE;


  public boolean setCam(Point3d eye, Point3d target)
  {
    /*initiate camera by setting  eye(position of
      the camera) and target(the position where
      camera is staring at).*/
    lib.AU3_WinActivate(SKP_TITLE, "");
    lib.AU3_ControlFocus(RC_TITLE, "", "Edit1");

    lib.AU3_Send("anim.setCam(" + eye.x + "," +
        eye.y + "," + eye.z + "," +
        target.x + "," + target.y + "," + target.z +
        ")", 1);
    lib.AU3_Send("{ENTER}", 0);
    lib.AU3_Sleep(1000);
    skp.getDisplay();
    lib.AU3_ControlSetText(RC_TITLE, "", "Edit2",
        "");
    // clear the display
    lib.AU3_Sleep(1000);
    lib.AU3_WinActivate(EDITOR_TITLE, "");
    return true;
  }
```

Appendix C. API Document

Appendix C.1 Class Terrain

```
public class Terrain()
```

Terrain class contains a constructor for creating landscape in SketchUp and other methods for manipulating the landscape, adding scenery, creating camera, generating animations for models, and initiating animation. The Terrain class is the main class for constructing a virtual world and can be instantiated.

Constructor Summary

Constructor and Description
`Terrain(int depth, int resolution, int plgSize)` Construct terrain model from depth, resolution, and size of each polygon in the terrain.
`Terrain(String file, int plgSize)` Construct terrain model from given coordinates file and size of each polygon.
`Terrain(String file, int depth, int plgSize)` Construct terrain from given heightmap, depth of terrain, and size of each polygon. A resolution of terrain depends upon a resolution of the heightmap.

Method Summary

Return Type	Method and Description
`void`	`addModel(Model model)` Create 3D model in the scene relatively to data in Model instance.
`void`	`addShape(Shape shape)` Add shape to the scene including block and cylinder.
`void`	`enableShadow()` Toggle shadow in the scene.
`Step[]`	`genStepsArray(Point2f start, Point2f dest, double sAngle, double dAngle, int startTime, int stopTime)` Create steps array representing animation by specifying start, destination, rotations at start and destination, start time, and end time of the animation.
`Step[]`	`genStepsArray(Point2f start, Point2f dest, int startTime, int stopTime)` Create steps array representing animation from start to dest during startTime and stopTime with no rotation of model along the path.
`Step[]`	`genStepsArray(Point2f start, Point2f dest, double dAngle, int startTime, int stopTime)` Create steps array representing animation from start to dest during startTime and stopTime with rotation for dAngle along the path.

`ArrayList<Model>`	`getDynamicModels()` Return list of dynamic models.
`float`	`getHeight(float x, float y)` Return height Z of given coordinate on the terrain surface.
`Model`	`getModel(String name)` Return a model relatively to a given name.
`double`	`getResolution()` Return resolution of terrain model.
`ArrayList<Shape>`	`getShapes()` Return list of shapes added to the scene.
`ArrayList<Model>`	`getStaticModels()` Return list of static models.
`void`	`loadStatics(String fnm)` Create static models from data in the text file in following format: b block-name x y width length height rotation texture-fnm [roof-texture-fnm] c cylinder-name x y radius height rotation texture-fnm m model-fnm model-name x y scale rotation.
`void`	`setLevel(int level)` Define a number of texture levels in the terrain.
`void`	`setTexture(float num, String file, int height)` Specify texture and height of each texture level.
`void`	`setTexture(float num, String file)` Specify texture of flat terrain without height of texture level.
`void`	`startAnimation()` Initiate animation at time zero.
`void`	`startAnimationAt(int time)` Initiate animation at specific time, for collision recovery.
`void`	`texture()` Start texturing the terrain accordingly to the defined texture levels

Appendix C.2 Class SKPControl

```
public class SKPControl()
```

SketchUp class contains methods for initiating, communicating with SketchUp, and setting a listener for animation-related events.

Method Summary

Return Type	Method and Description
`boolean`	`startSketchUp()` Initiate SketchUp, create a new project, and start Ruby console.
`boolean`	`closeWin(String title)` Close a window accordingly to a given title name.

`String[]`	`getDisplay()` Return the Edit2 display text as an array of lines.
`void`	`setDirectory(String directory)` Set a project directory path.
`String`	`getDirectory()` Return a project directory path.
`void`	`setListener(AnimationListener lis)` Set animation listerner.

Appendix C.3 Class Model

```
public class Model()
```

Model class provides constructor for creating static and dynamic scenery models. The class contains methods for loading steps array to the model on SketchUp side, setting, and getting model's parameters. The Model class can be instantiated.

Constructor Summary

Constructor and Description
`Model(String file, Point2f position, int offsetZ, float scale, float angle, boolean movable)` Create a model on SketchUp from model file name, position, offset in Z axis, scale, and rotation. The last argument defines whether the model is static or dynamic.
`Model(String file, Point2f position, float scale, float angle, boolean movable)` Create a model on SketchUp side without requiring offset in Z axis.

Method Summary

Return Type	Method and Description
`void`	`setName(String name)` Define a model name, if preferred.
`String`	`getName()` Return a model name.
`void`	`setPosition(Point2f position)` Define a model's 2D position.
`Point2f`	`getPosition()` Return model position.
`void`	`setZ(float height)` Set a position of the model in Z axis.
`float`	`getZ()` Return a position of model in Z axis.

`void`	`setRotation(float angle)` Define rotation of model.
`float`	`getRotation()` Return rotation of the model.
`void`	`setScale(float scale)` Define model scale.
`float`	`getScale()` Return model scale.
`void`	`setWidth(int width)` Define model’s width.
`int`	`getWidth()` Return model’s width.
`void`	`setLength(int length)` Define model length.
`int`	`getLength()` Return model length.
`void`	`setHeight(int height)` Define height of the model in Z axis.
`int`	`getHeight()` Return height of the model.
`void`	`setFile(String file)` Set the model file name (the model file must stored in SkethUp\Components\Objects).
`String`	`getFile()` Return the model file name.
`void`	`setDynamic(boolean movable)` Specify whether the model is dynamic.
`boolean`	`isMovable()` Return dynamic status of the model.
`void`	`setOffsetZ(int offsetZ)` Set model offset in Z axis.
`int`	`getOffsetZ()` Return Z offset of the model.
`void`	`setAnimation(Step[] stepsArray)` Write the data in steps array to the model’s steps file
`Step[]`	`getAnimation()` Return steps array of the model.
`Step`	`getAnimation(int index)` Return a step from steps array of the model accordingly to the given index.
`boolean`	`loadAnimation()` load the data in model steps file to SketchUp and create steps array of the model on SketchUp side.

Appendix C.4 Class User

```
public class User extends Model()
```

User class extends a Model class and can be instantiated. The class contains a constructor for the user model which automatically defines the name of the model and a method for attaching the camera to the user model.

Constructor Summary

Constructor and Description
`User(String file, Point2f position, int offsetZ, float scale, float rotation)` Construct the user model from file name, position, Z offset, scale, rotation, and set its name to 'User'.

Method Summary

Return Type	Method and Description
`boolean`	`setCam(Point3d eye)` Set the camera to look at the user model. Only the position of the camera is required and the camera follows the user model as it is animating.

Appendix C.5 Class Shape

```
public class Shape extends Model()
```

Shape class extends the Model class. The class contains methods for getting, setting shape parameters, and texturing a shape. The class is instantiated by its subclasses; Block and Cylinder.

Method Summary

Return Type	Method and Description
`void`	`setRadius(int radius)` Define radius of the shape.
`int`	`getRadius()` Return radius of the shape.
`void`	`setType(String type)`

	Define type of the shape (block or cylinder).
`String`	`getType()` Return type of the shape.
`boolean`	`texture(String sideTex)` Apply sideTex to the side of shape and black color to the roof.
`boolean`	`texture(String roof, String side) int startTime, int stopTime)` Apply side texture and roof texture to a shape.

Appendix C.6 Class Cylinder

```
public class Cylinder extends Shape()
```

Cylinder class provides constructors for creating cylinder in SketchUp with and without offsetZ. The class extends Shape class and can be instantiated.

Constructor Summary

Constructor and Description
`Cylinder(Point2f position, int offsetZ, int radius, int height)` Construct plain cylinder model on SketchUp side from position, offsetZ, radius and height.
`Cylinder(Point2f position, int radius, int height)` Construct plain cylinder model on SketchUp side from position, radius and height.

Appendix C.7 Class Block

```
public class Block extends Shape()
```

Block class contains constructors for creating block in SketchUp with and without offsetZ. The class extends Shape class and can be instantiated.

Constructor Summary

Constructor and Description
`Block(Point2f position, int offsetZ, int width, int length, int height)` Construct block model from position, offsetZ, and block dimension(width, length, and height).

`Block(Point2f position, int width, int length, int height)` Construct block model from position, and dimension(width, length, and height).

Appendix C.8 Class CollisionEvent

```
public class CollisionEvent()
```

CollisionEvent class provides constructor for creating collision event instance and get methods for accessing the collision data. The class can be instantiated.

Constructor Summary

Constructor and Description
`CollisionEvent(Object source, String file)` Construct CollisionEvent instance which stores the data in collision file

Method Summary

Return Type	Method and Description
`String`	`getSourceName()` Return the name of collision source model
`String`	`getTargetName()` Return the name of collision target model
`Point2f`	`getSourcePos()` Return 2D position of collision source
`Point2f`	`getTargetPos()` Return 2D position of collision target
`Point2f`	`getSourceDimen()` Return 2D dimension of collision source model
`Point2f`	`getTargetDimen()` Return 2D dimension of collision target model
`float`	`getSourceAngle()` Return rotation of collision source model
`float`	`getTargetAngle()` Return rotation of collision target model
`int`	`getTime()` Return collision time
`Step[]`	`getSourceStep()` Return steps array of source
`Step[]`	`getTargetStep()` Return steps array of target

Step	getSourceStepAt(int time) Return a step from source’s steps array at specific time
Step	getTargetStepAt(int time) Return a step from target’s steps array at specific time

Appendix C.9 Class CollisionRecovery

`public class CollisionRecovery()`

CollisionRecovery class contains a method for automatically solve a dynamic collision and other support methods to help implementing collision recovery.

Method Summary

Return Type	Method and Description
void	basicRecovery(TerrainConstructor terrain, CollisionEvent e) Use collision information to calculate coordinate P1, P2, P3, and P4. Then update steps array of source and target in order to make source model moves to P1-P4(around the target), while the target is waiting.
Step[]	extractArray(int index, int time, Step[] stepsArray) Return subarray from stepsArray started from the given index and update the time of each element of the array started from 'time'.
int	findClosestPoint(Point2f p4, Step[] stepsArray) Return the index of position closest to given point p4
boolean	stopModel(String name) Stop the animation of the model accordingly to the model name.

Appendix C.10 Class Scene

`public class Scene()`

Scene class is a super class for the user Java program. it contains a method for setting up the camera to the scene.

Method Summary

Return Type	Method and Description
boolean	setCam(Point3d eye, Point3d target) initiate camera by setting eye(position of the camera) and target (the position where camera is staring at).

Appendix D. Photocopy of the Paper

The 7th International Conference on Engineering and Technology
ICET-2015, Phuket, June 19-20, 2015
Prince of Songkla University, Faculty of Engineering
Hat Yai, Songkhla, Thailand 90112

ANIMATED 3D VIRTUAL WORLDS USING JAVA AND SKETCHUP

Chonmaphat Roonnapak, Andrew Davison
Dept. of Computer Eng. Faculty of Eng., Prince of Songkla University, Hat Yai, Thailand
Email: 5610120049@email.psu.ac.th, ad@fivedots.coe.psu.ac.th

Abstract: *This paper introduces a Java API for creating animated 3D virtual worlds in SketchUp, offering a feature set aimed at novice graphics programmers. The API provides methods for generating a landscape, loading (and creating) scenery, camera mobility, and discrete-event based animation of models. The API supports a listener interface, which supplies information about changing events in the world, such as collisions between models.*

Key Words: ***3D virtual worlds/animation/ SketchUp/ discrete time events***

1. INTRODUCTION

There are many excellent tools for constructing 3D scenes, including 3D Max, Unity 3D, AutoCAD, IMAGIS and SketchUp (http://www.sketchup.com/). SketchUp has several advantages for users new to 3D graphics: simple modeling, free online materials, and a Ruby console interface [1]. Java programmers can choose from several 3D APIs e.g. Java 3D, JOGL, and JMonkey Engine. However, they are designed for general 3D programming, such as games development, and most are OpenGL-based, making them difficult for novices to understand and apply.

Our Java API focuses on virtual world creation for programmers new to 3D graphics by utilizing SketchUp. It supports landscape generation, dynamic and static models, camera mobility, animation, and high-level event handling. A typical scene is shown in Figure 1.

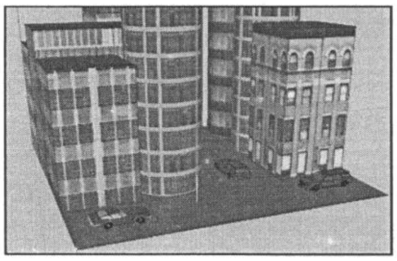


Fig. 1. Example 3D virtual world

2. SYSTEM OVERVIEW

The system consists of two parts, shown in Figure 2. A programmer utilizes the Java API for defining a 3D scene, which is rendered and animated by SketchUp.

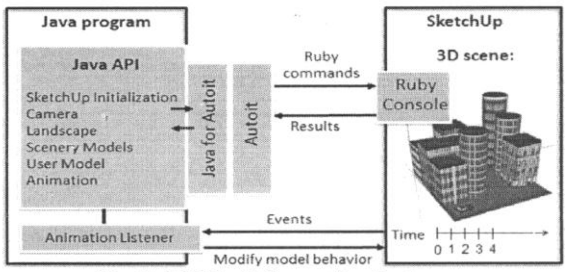


Fig. 2. 3D World Generating System

SketchUp's Ruby interface is based around the user typing commands into an on-screen console [2], rather than through an exported API. This makes it rather difficult to interface an external program with SketchUp.

Our solution is to utilize AutoIt (https://www.autoitscript.com/site/autoit/), a popular freeware scripting language for automating on-screen actions involving the keyboard and mouse (e.g. clicking on a close box to terminate an application). We call AutoIt through a Java interface to send commands to the SketchUp-Ruby console (http://fivedots.coe.psu.ac.th/~ad/jau/).

Another problem is that SketchUp does not issue events when things change inside its 3D scene. We implement this functionality by having SketchUp modify and create various data files; any modifications to these files are detected by a Java thread running inside our API. This low-level file manipulation and monitoring is hidden from the API programmer, who is notified of events via a listener interface. For example, when two models collide in the 3D scene, SketchUp creates a new file. The Java thread detects the file creation, reads information about the event from the file, and calls a suitable event handling method in the programmer's code.

3. VIRTUAL WORLD GENERATION

A typical virtual world is composed of a landscape, various static scenery models, dynamically moving models, a model representing the user, and a camera. This section briefly describes how the programmer can generate these elements

A landscape is created by employing the Diamond-Square algorithm [3], which is controlled through a simple interface suitable for graphics programming beginners, while still being capable of achieving interesting results.

Terrain texturing is based on landscape height ranges, with the API automatically generating blended texture transitions between those ranges.

One of the advantages of using SketchUp is the availability of free or low-priced models (e.g. from the 3D Warehouse, https://3dwarehouse.sketchup.com/). The programmer must specify a model's (x, y) position, z-axis orientation, scale, and whether the model will be static or dynamic. The API calculates the model's z position by assuming that all models rest on top of the terrain.

We include support for textured block and cylinder shape creation, which is useful when creating city-like scenes (several examples can be seen in Figure 1),

The API includes a 'user' model, a dynamic model representing the user, which can be automatically followed by the camera as the model moves.

4. ANIMATION

Each dynamic model is assigned a "steps" array which determines its motion during a specified period. Each animation begins and ends relative to SketchUp's global time, as illustrated in Figure 3. Car1 starts moving at time 1 while Car2 starts at time 5.

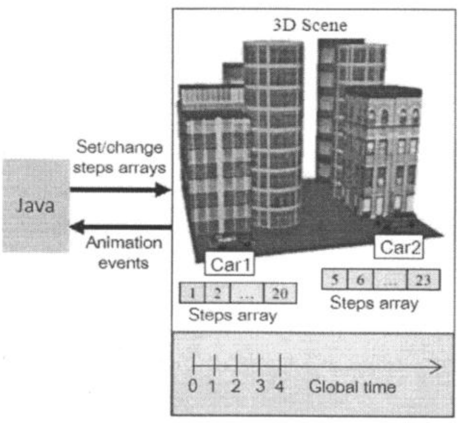


Fig. 3. The Animation Behavior

If an animation-related event occurs, such as a collision between models, the Java programmer will be notified via a collision event, and can then choose to modify the steps arrays of the models involved in the collision.

This section examines five aspects of our animation support in more detail: movement generation (i.e. how a steps array is created), how steps arrays can be changed at run time, collision detection, listening for events on the Java side, and collision recovery.

4.1 Movement Generation

On the SketchUp side, each model is animated via the execution of its steps array. An example appears in Figure 4.

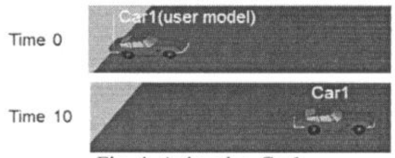


Fig. 4. Animating Car1

The programmer supplies information about a model's position and orientation at the start of the animation, and its final position and orientation. For instance, the car in Figure 4 starts moving at time 1, and finishes at time 20. During that period, it will move 170 units along the x-axis with no change in its orientation (i.e. it remains pointing to the right). A suitable steps array is created by calling Movement.genMovement():

```
Step[] steps1 = mover.genMovements(
  landscape,
  new Point2f(10, 100),  // start pos
  new Point2f(180, 100), // stop pos
  0, 0,       // start/end rotations
  1, 20);     // start/stop times
```

The programmer only has to supply (x, y) positions for the model. The API assumes that it will move over the surface of the specified landscape, and calculates its z-positions accordingly, using a variation of the standard Barycentric approach [4]. When the vertices of the current landscape triangle are known, the z-position of any (x, y) position in the triangle can be computed

However, a problem occurs if the steps array contains two points (e.g. S0 and S1) which cross a boundary between two terrain triangles. If the terrain is rough, the orientation of the triangles may not be in the same plane, as in Figure 5. If the model moves directly between the two points, then it will disappear beneath the landscape (the dotted line in Figure 5). An extra coordinate at the intersecting edge of the triangles must be added to the steps array (i.e. S01 in Figure 5).

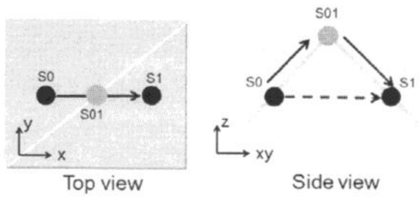


Fig. 5. Extra Coordinate on the Intersected Edge

4.2 Animating a Model with a Steps Array

The API allows a model to be assigned multiple animation sequences. Therefore, it's quite likely that the resulting steps array will not be contiguous, as in Figure 6.

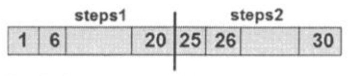


Fig. 6. Steps Array with a Waiting Period

The first part of the array defines an animation between times 1 and 20, while the second commences at time 25 and ends at time 30. These are processed by having the model waits for 4 seconds during the times 21-24.

Another issue is when the times of two animations for a model overlap in time. The API handles this by ignoring the overlapping period of the latest animation, while issuing a warning to the programmer. The following pseudo code demonstrates the behavior of the animation processing loop on the SketchUp side:

```
max = max stop time of all models
 0...max {|i|   // global time of the scene
  for each model, m
    if i ≥ start time of m
     animate using m's steps array at time i
     or do nothing if no value at time i
}
```

The main problem with this pseudo code is that it does not deal with collisions, which are covered in the next sections.

4.3 Collision Detection

There are two types of collision detection: *dynamic* (between two moving models) and *static* (when a moving model hits a piece of scenery, such as a building).

Assume that Car1 is about to move from position (x1, y1) to (x2, x2) at time 5. The mechanism for dynamic collision detection between Car1 and another model, Car2, can be explained by reference to Figure 7.

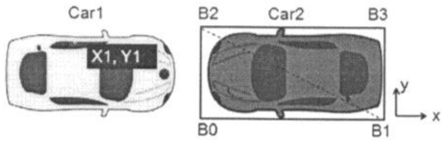


Fig. 7. Dynamic Collision Detection

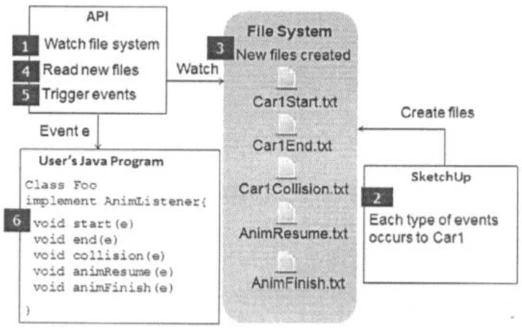
The boundary of Car2 is split into two triangles. If (x2, y2) falls inside one of these, then a collision will occur. The model's movement is temporarily stopped, and a new file of collision information is created. The animation loop's pseudo code can be modified to include this behavior:

```
max = max stop time of all models
0...max {|i|   // global time of the scene
  for each model, m
    if i ≥ start time of m
     if isCollision(m, nextPosition)
       stop m moving;
       create a collision file for m
    else
       animate by m's step array at time i
       or do nothing if no value at time i
  }
```

Static collision detection (between moving and stationary models) uses a similar algorithm, but is performed outside the time-critical SketchUp animation loop, during the generation of a steps array. As each coordinate is calculated, the collision detection test is carried out to determine if the point falls inside any of the static models. The API raises an exception to inform the programmer to change their animation sequence.

4.4 Animation Listener

The Java-side animation listener allows a programmer to handle a range of collision-related and animation events. Its underlying file-based mechanism, which is hidden from the programmer, is shown in Figure 8.



```
Class Foo
implement AnimListener{
 void start(e)
 void end(e)
 void collision(e)
 void animResume(e)
 void animFinish(e)
}
```

Fig. 8: Animation Listener Mechanism

Our API employs this approach due to the communication limitations of SketchUp's Ruby API.

In stage 1 of Figure 8, a Java thread is created to act as a file system monitor. When an important event, such as a collision, occurs inside SketchUp (stage 2), a new file is created (or an existing one modified) in stage 3. This change is detected on the Java side (stage 4), and the thread uses the file's information to create a Java event (stage 5), which is passed to the programmer's AnimListener interface (stage 6) where a relevant method is called.

Currently our API supports five event types: StartEvent, EndEvent, ResumeEvent, FinishEvent, and CollisionEvent. Each event is handled by a different method in the AnimListener interface. For example, CollisionEvent objects are passed to the collision() method.

4.5 Collision Recovery

The API's design allows the programmer to define their own collision recovery behavior inside the AnimListener collision() method. Since such behaviors are rather complicated, the API offers a good default behavior, implemented as the basicRecovery() method. This allows collision recovery to be defined as:

```
public void collision(CollisionEvent e)
{ CollisionRecovery.basicRecovery(
                          landscape, e); }
```

The CollisionEvent object contains a variety of information, including the collision model's name, dimensions, the other model's name, the collision position, and time.

basicRecovery() utilizes the landscape and the collision event object. The recovery approach is presented in Figure 9.

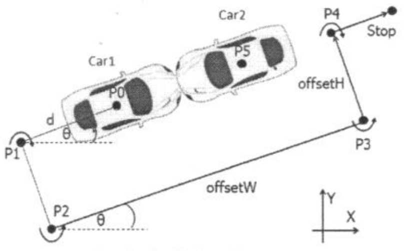


Fig. 9: Collision Recovery

Assume that Car1 is currently being animating and a potential collision with Car2 is detected. basicRecovery() calculates four positions (P1, P2, P3 and P4 in Figure 9) and updates Car1's steps array with an animation sequence that passes through those points. The calculation of the coordinates is described by the following equations:

```
d = Car1.width
offsetH = Car1.height + Car2.height
offsetW = Car1.width + Car2.width + d
P1 = (P0.x - d*cos(θ), P0.y - d*sin(θ))
P2 = (P1.x + offsetH*cos(90-θ),
      P1.y - offsetH*sin(90-θ))
P3 = (P2.x + offsetW*cos(θ),
      P2.y + offsetW*sin(θ))
P4 = (P1.x + offsetW*cos(θ),
      P2.1 + offsetW*sin(θ))
```

Car2's animation behavior is also modified: it remains stationary for a short time, and then resumes its steps array sequence.

For example, assume that the collision happens at time 10. The steps array for Car1 is updated so that the model moves around Car2. Assuming that the animation through points P0-P4 takes five seconds, then a new array is generated as shown in Figure 10.

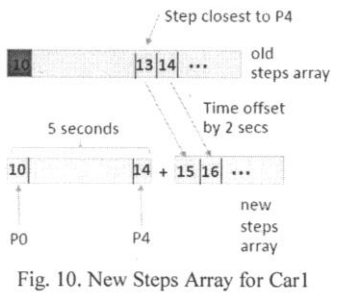


Fig. 10. New Steps Array for Car1

The steps array is modified in two ways. A new sequence is added extending from times 10 to 14 which moves the model from point P0 to P4. The old step array is discarded up to time 13 which is determined to be the position in the old animation sequence closest to P4. The sequence starting at time 13 is then time-displaced by 2 units so that it continues after the car has reached P4.

The generation of the P0-to-P4 animation sequence may be more complicated than this since the collision recovery path may itself collide with a static model, in which case it will need to be modified. In the worst case, basicRecovery() may be unable to modify the model's steps array and so an exception is raised.

The model involved in the collision (Car2 in this example) waits for three seconds before continuing on its previous route. The new steps array for Car2 is shown in Figure 11.

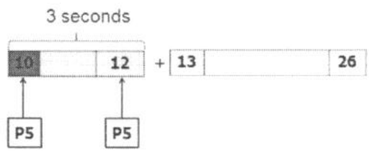


Fig. 11. New Steps Array for Car2

The collision position of the car (P5) remain unchanged from time 10 to 12, and then the old steps array sequence that started at time 11 is resumed at time 13, after being time-displaced by 2 units. After the steps arrays of Car1 and Car2 are updated and reloaded, SketchUp will resume the animation starting from the collision time (time 10 in this example).

5. CONCLUSIONS

Existing 3D libraries for Java are designed for general needs too complicated for novice graphics programmers. Our simpler API can generate complex, visually pleasing, virtual worlds in SketchUp and supports robust discrete-time event-based animation, with collision detection and recovery.

Future work will focus on extending the event model, adding skeletal animation to the user model, improving the dynamic camera behavior, and supporting lighting variations.

REFERENCES

[1] Tal, D. 2009. "Google SketchUp for Site Design: A Guide to Modeling Site Plans, Terrain and Architecture", Wiley.

[2] Scarpino, M. 2010. "Automatic SketchUp: Creating 3-D Models in Ruby", Eclipse Engineering LLC, Hanover.

[3] Kui-Ru, L. and Xiao-Feng, L. 2010. "An Accumulated Method Based on Fractal for Automatic Terrain Generation", In *E- Health Networking, Digital Ecosystems and Technologies (EDT), 2010 International Conference* (Shenzhen Apr 17- 18), IEEE, Piscataway, N.J., 426-430.

[4] Dunn, F. and Parberry, I. 2002. "3D Math Primer for Graphics and Game Development", Wordware, 260-267.

VITAE

Name Mr. Chonmaphat Roonnapak
Student ID 5610120049
Educational Attainment

Degree	Name of Institution	Year of Graduation
Bachelor of Computer Engineering	Prince of Songkla University	2015

List of Publication and Proceeding

Roonnapak C., Davison A, "Animated 3D Virtual Worlds Using Java and SketchUp", in The 7th PSU-UNS International Conference on Engineering and Technology, Phuket, Thailand, 2015.