České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Jakub Petriska**

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: **Rozšiřitelná knihovna herních mechanizmů pro Android**

Pokyny pro vypracování:

Navrhněte a implementujte základní koncept knihovny 3D herních mechanizmů (3D game engine) pro platformu Android. Soustřeďte se především na návrh efektivní architektury knihovny, grafické možnosti mohou být základní (implementujte vlastní základní OpenGL renderer). Knihovna bude umožňovat zobrazování primitivních objektů a importování modelů ze souboru (např. obj). Pohyb s objekty a úprava jejich vlastností za chodu bude realizována pomocí skriptů v jazyce Java. Implementujte detekci kolizí objektů.
Knihovna bude snadno rozšiřitelná o další nezávislou funkcionalitu, jako např. o nástroje pro analýzu nebo o využití standradního Android UI.
Pomocí knihovny vytvořte fungující prototyp hry (proof of concept), na kterém demonstrujete implementovanou funkcionalitu.

Seznam odborné literatury:

Bc. Martin Štýs, Komponentový přístup při tvorbě objektů herní knihovny 2014, Diplomová práce ČVUT FEL

Jesse Schell, The Art of Game Design: A Book of Lenses, CRC Press 2008

Mike McShaffry and David Graham, Game Coding Complete 4th edition, Course Technology Cengage Learning, 2013

3D Game Engine Design : A Practical Approach to Real-Time Computer Graphics, David H. Eberly, Morgan Kaufmann, 2000

Collision Detection in Interactive 3D Environments, Gino van den Bergen, Morgan Kaufmann, 2003

Vedoucí: Ing. David Sedláček, Ph.D.

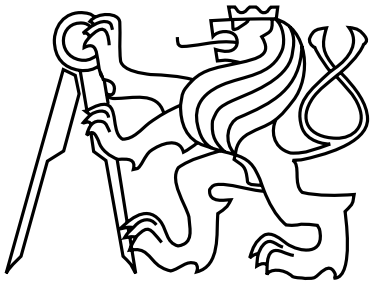
Platnost zadání: do konce letního semestru 2015/2016

doc. Ing. Filip Zelezný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 26. 3. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Bachelor's thesis

Extensible game engine for Android

Jakub Petriska

Supervisor: Ing. David Sedláček, Ph.D.

Study Programme: Software technologies and management

Field of Study: Software engineering

May 20, 2015

Acknowledgements

I would like to thank Ing. David Sedláček, Ph.D. for supervising and supporting my work and Tereza Švarcbachová for helping with corrections.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 22, 2008 .....

Abstract

This thesis covers design and implementation of a simple game engine for Android platform and describes how it is used. Project is aimed at creating simple-to-use, genre independent and extensible game engine. Main focus is placed on maintainable and extensible architecture and simple scripting API, while only providing primitive rendering capabilities.

Abstrakt

Tato práce zahrnuje návrh a implementaci herního enginu pro platformu Android a popisuje způsob jeho použití. Projekt je zaměřen na vytvoření jednoduše použitelného, žánrově nezávislého a rozšiřitelného herního enginu. Hlavní důraz je kladen na udržitelnou a rozšiřitelnou architekturu a jednoduché skriptovací API, zatímco grafické možnosti zůstávají pouze základní.

Contents

Chapter 1

Introduction

Nowadays game industry is huge and is still growing. Gaming competitions, so called eSports, are coming close to the usual sports events in terms of viewers and prize money[1]. Gaming on mobile devices is no exception. Every now and then there is a game that breaks the records in terms of sales and revenue. Notable example of this is a game called Crossy Road released on November 2014 earning 3 million dollars through video advertisment[2]. Mobile games can also provide long-lasting source of income. Best example is Finnish mobile games developer Supercell with their hit Clash of Clans. Supercell's revenue in 2014 reached 1.7 billion dollars[3].

Game engines are key part of this industry providing environment for faster development of games. While there are private game engines used only by their developers and perhaps a handful of others, there are also game engines meant to be widely used by lots of developers for games of various genres. Examples of such engines are Unity engine[4] and Unreal engine[5]. These engines allow developers to create a wide variety of games where genre is not limited by the engine but only by developer's imagination. As of March 2015 both of the engines are mostly free and even before both of them were very affordable. Low price of such tools allows easy entry into the game industry even for developers without great resources. In the mobile waters the entry can be even easier since many mobile mobile games are simple in comparison with desktop titles. Engines also thrive to make using the engine easy even for developers without huge experience in programming. This simplifies game development mainly for newcomers.

Both mentioned engines and tons of other engines provide solutions for many projects. While there may not be a need for new engine, the skill and experience with making games

[1] <http://www.polygon.com/2014/7/19/5918845/dota-2-international-2014-champions-money-winners> (May 18, 2015)
[2] <http://venturebeat.com/2015/03/03/crossy-road-earns-3m-in-revenue-from-unitys-video-ads/> (May 18, 2015)
[3] <http://venturebeat.com/2015/03/24/clash-of-clans-developer-supercells-revenues-tripled-in-2014/> (May 18, 2015)
[4] <http://unity3d.com/> (May 18, 2015)
[5] <https://www.unrealengine.com/> (May 18, 2015)

and game engines is very desired and for me personally very interesting. Therefore in this thesis I will design and implement a simple three dimensional (3D) game engine for Android platform. Project will focus on genre independence and ease of use while setting graphical capabilities aside. I will describe engine's design, show how engine is used and create and present a showcase game using the engine. In addition to the experience, this project will bring a platform which can be further expanded and can serve for easy experimenting with various interesting techniques and technologies used in the game development.

Chapter 2

Vision

In this chapter I present some high-level goals and concepts that the engine should fulfill and implement. Holding on to these concepts will ensure that the engine will be on the right track to become a successful product in case the work on it continues and enough features are added. Detailed discussion about more aspects of games and game development can be found in book by Schell (2008) [5].

Usability By making the engine easy to use potential user base can be expanded by newcomers into the field of games development. Fast learning can ensure a lot of potential users will try the engine.

Fast prototyping Allowing user to quickly build a game prototype is very important especially in the field of mobile games. Games for mobile platforms are very often quite simple, however not all game ideas make sense when implemented. Hence making prototypes in short time is crucial.

Genre independence Fitting the engine on a single game genre vastly reduces the possibilities for produced games. Most importantly it disallows creation of new game genres.

Expandability In this case expandability means allowing users to develop additional engine functionality. It should also be possible to easily distribute this functionality among other users. This can significantly help engine users since development of particular functionality doesn't always have to be done by engine developer but can also be done by engine user and than distributed among users who need this functionality.

Ability to communicate with underlying platform Communication will allow sending data out of the engine. Data can then be displayed or interpreted as commands and a reaction can follow. This enables connecting the engine with any platform dependent functionality. One of the examples is using user interface (UI) framework from Android SDK while running on Android device.

With platform independence in mind Possibility of creating a game once and then deploying to different platforms is huge time saver in today's development. In this thesis I am developing the engine only for Android but I will design it in such a way

that in future it will be possible at least to bring some of the implemented concepts into truly multiplatform engine.

Chapter 3

Design

I chose to write the engine in the Java language. I was deciding between C++ with the use of Native Development Kit for Android and Java. I chose Java because of the faster implementation it enables, which was much needed because of my little experience with creation of game engines. The engine supports Android from version 2.3 and up. Android 2.2 adds support for OpenGL ES 2.0. OpenGL ES API significantly changes between 1.0 and 2.0 versions making these versions incompatible. However code remains compatible from version 2.0 to 3.0[1]. Another restriction is that vertex buffer objects (VBO) support was added in Android 2.3. VBOs provide performance gain since they allow programmer to upload data to graphics processing unit (GPU) once and later use this already uploaded data instead of uploading it in every step of game loop. Due to this I decided to start with OpenGL ES version 2.0 and Android 2.3.

During this project I developed two versions of the engine. First version used 3rd party rendering library called jPCT[2] and implemented basic scripting. However later I decided to start from scratch again and implement better second version. This new and current version uses it's own renderer instead of jPCT library. Reason for custom renderer was that renderer implemented specifically for this engine can be done in such a way that allows easier integration instead of forcing an existing library into the engine. This later turned out to be huge advantage since the new renderer fits in naturally while the 3rd party library didn't. I attribute this to the fact that jPCT library already used a concept of object in the game world which was also present in the engine itself.

I decided to use a left handed coordinate system with X axis pointing to the right, Y axis pointing up and Z axis pointing forward. Engine API will also contain its own vector and matrix classes for completeness. These will be used by the engine itself and they can also be used by game programmers. Therefore these classes will implement most of the operations expected from such classes. Dunn & Parberry (2011) [3] provide details about operations with vectors and matrices. Relative rotation stored on game objects will be stored as Euler angles and later converted into complete and absolute transformation matrix. More about

[1] <http://developer.android.com/guide/topics/graphics/opengl.html#compatibility> (May 18, 2015)
[2] <http://www.jpct.net/> (May 18, 2015)

Euler angles and transformations can be also found in work of Dunn & Parberry (2011) [3] or a book by Eberly (2000) [1].

3.1 Inner workings

Core of engine's operation is tree structure of so called game objects. Game objects provide position, orientation and scale in world's space. Every game object holds its relative transformation and its absolute world transformation is composed from its own transformation and transformations of all of its parents. Game objects also hold another very important thing, the components. Components provide functionality to game objects. This means they can render a 3D object in world or detect collisions. In every iteration of game loop engine walks through this tree of game objects and components and calls various life cycle methods of the components. These methods allow components to do such things as initialize themselves, update their state or react to the updated state of other components or game object itself. More about game objects and components and how they are represented in engine's code will be explained in the chapter about scripting API. Concept of holding the functionality in components and different variations of this approach are discussed in detail in bachelor thesis from Martin Štýs (2014) [6].

Now it's a good time to mention how scripting fits into the whole concept. Scripting allows the engine user, game programmer, to create behavior of his game. This mostly consists of changing the state of objects in the game world but can also do other things not directly related to gameplay. To create a script user creates a component. This means that user scripting fits into the engine the same way its default functionality such as rendering does.

Another question to answer is how do game objects with components get into the game world in the first place. This is done through a XML file which defines the objects in the scene along with their components. This file allows you to specify the hierarchy of game objects, their parent-child relationships, their position, rotation and scale and allows you to add components to them and set various parameters on the components. This way you configure the whole initial state of the game scene from which your game plays. This approach is inspired by Graham & Schaffry (2013) [4] which describes a component architecture of a game as well as defining scene using game objects and components in XML files. Engine also supports having multiple scenes that can be switched as needed.

3.2 Access to engine's functionality

To create simple and easy way to give access to all the functionality the engine offers, it is kept in one place. This place is an instance of *com.monolith.api.Application* class which has methods that return other objects providing specific functionalities such as touch input or time information. This object also has methods to switch between game scenes. Reference to this instance is held in every game object and every component making it easily accessible.

3.3 Rendering

Rendering is on very basic level. *Application* object provides the *Renderer* which has methods that render meshes with specific world transformation. In current implementation rendering uses shader with one directional light with constant color. This light so far cannot be changed in any way. In terms of meshes engine provides one primitive object which is a cube and allows import of .obj files to load mesh geometry and normals.

3.4 Touch input

Touch input data are provided by a middle man object which serves as adapter between platform's native touch input and engine's unified way of providing touch input. It's role is to listen for and store all touch display interaction during one game loop step and present it for further processing by game code during the next game loop step. This kind of storing events by step is important since in the Android UI framework reaction to touch events can be done any time, but in the engine it can only be done in specific time meant for update of game state.

3.5 Collision detection

In the thesis I will refer to objects whose collisions are detected as colliders. This term is adopted from Unity engine which uses the similar wording.

Main part of the collision detection is the so called *CollisionSystem*. Instance of this system can be obtained through *Application* object. This system provides methods for registering and unregistering colliders. When state of this system is updated it checks for collisions between all registered colliders. The topic of listening for collisions in game code will be discussed in the next chapter. Current engine implementation only supports one type of collider, a arbitrarily oriented box the so called oriented bounding box (OBB).

The collision of two OBBs is implemented using the separation axis theorem. The implementation is taken from a book by Ericson (2004) [2]. Algorithm checks for collision between every pair of bounding volumes that it has registered. To optimize the detection, colliders can also be divided into different groups. Colliders in the same group are not checked for collisions. This can vastly decrease a number of collision checks since for example all non movable objects can be put into one group and collisions between them are not checked.

3.6 Engine integration into applications

Whole engine can be started by creating an instance of the *com.monolith.engine.Engine* class that encompasses the whole engine. This object takes care of everything that the engine does. However, to start it the creator must provide implementations of certain objects that are platform dependent. These objects provide such functionality as rendering or touch input and must be implemented and created by the platform and passed into the *Engine*.

3.7 Structure

Figure 3.1 shows class diagram of the engine. At the top of diagram the green box marks the classes specific to Android. Rest of the objects are from platform independent part. The *Engine* object encapsulates whole engine. All objects containing important features, such as rendering, collision detection or touch input, are depicted on its left side. Below is the *Application* which gains access to these objects through the *Engine*. *Application* object is in turn held by every *Component*. The *GameObject* in this diagram represents the game object tree held by *Engine*.

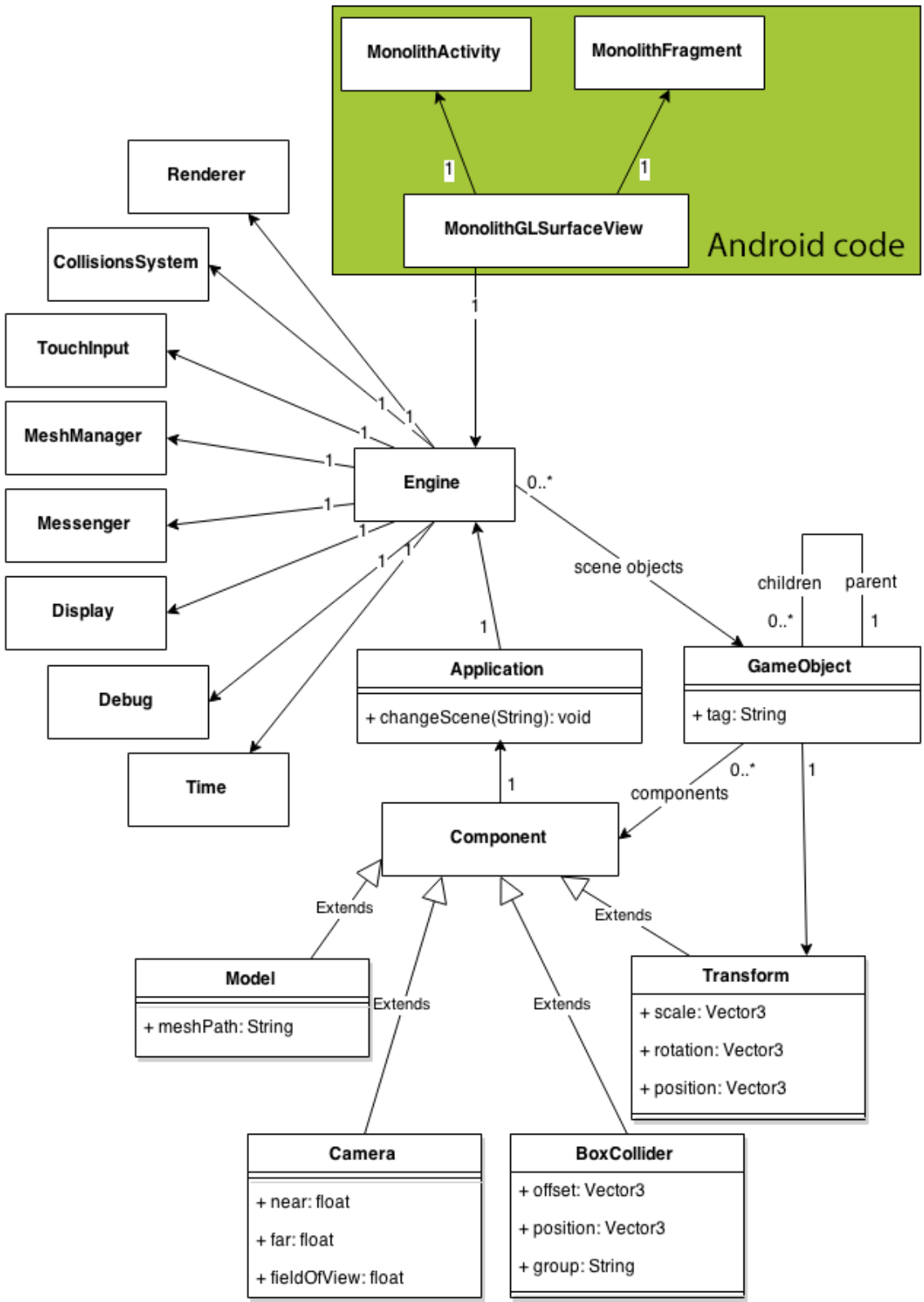


Figure 3.1: Structure of the engine

Chapter 4

Using the engine in Android application

4.1 Putting the engine into Android application

For Android the engine works as a library so the library files need to be added the standard way to the Android application. To show content rendered by the engine in the application the *MonolithActivity* or *MonolithFragment* must be used. These are subclasses of Android's own *Activity*[1] and *Fragment*[2] classes. The *Activity* represents one screen in the application so *MonolithActivity* shows a screen only with engine's content and Android's status bar and navigation bar[3]. *Fragment* is embedded into an *Activity* so *MonolithFragment* makes it possible to embed the content displayed by the engine into a custom *Activity* and display other content alongside the one from engine.

4.2 Scenes definition

To display any content, two XML files need to be created. On Android the engine has a special place in application where its files are stored. This place is in the *monolith* folder inside the *assets* folder. Everything used by the engine needs to be saved only to this folder. The first XML file is definition of our scene. It defines objects in our scene, their parent-child relationships and their components. Following example shows such a file.

[1] <https://developer.android.com/reference/android/app/Activity.html> (May 18, 2015)
[2] <https://developer.android.com/reference/android/support/v4/app/Fragment.html> (May 18, 2015)
[3] <https://developer.android.com/design/handhelds/index.html#system-bars> (May 18, 2015)

```
<scene>
    <gameObjects>
        <gameObject>
            <transform>
                <position x="1" y="0" z="2" />
                <rotation x="0" y="45" z="0" />
                <scale x="1" y="2" z="1" />
            </transform>
            <children>
                <gameObject>
                    <transform>
                        <position x="1" y="1" z="1" />
                    </transform>
                    <components>
                        <component type="camera">
                            <param name="far">1000</param>
                        </component>
                    </components>
                </gameObject>
            </children>
        </gameObject>
    </gameObjects>
</scene>
```

In the scene defined by this file there are two objects and one is a child of the other. All objects of the scene must be placed into *gameObjects* element which is placed in the *scene* element, the root of the document. To define a game object a *gameObject* element is used. The *gameObject* element also has a *transform* containing game object's relative transformation, which is optional or does not have to contain all information. In case *transform* or its parts are missing the default values of zero position, zero rotation and scale of 1 are used instead. Other elements in *gameObject* are *children* element containing its children game objects and a *components* element containing its components. Both of these are optional and can be left out instead of writing just empty element.

Components are contained within the *components* element in the *gameObject*. Type of component is specified by the attribute *type* of *component* element. Individual types of components that are built into the engine will be described in following chapters. Element *component* can also contain zero or more *param* elements. These specify the values of fields that are set on the component instance which is created during scene initialization. Only the parameters specified here are set otherwise the default values are used. The *name* attribute of *param* element states the name of the field in the component's class. Type of parameter can be any of the Java's primitive types or their respective wrapper classes.

To be able to launch the scene the second XML file needs to be created. This file must have exact name *scenes.xml* and must be placed in the root of engine's files folder, the

assets/monolith. This file defines what scenes our engine can show, their names and paths to their definition files. Following example shows the file's structure.

```
<scenes
    defaultSceneName="main_scene">
    <scene
        name="main_scene"
        sceneFilePath="scenes/main_scene.xml"/>
</scenes>
```

The available scenes are listed as *scene* elements in the *scenes* element. The *name* attribute of *scene* defines the name through which this scene will be referenced and the *sceneFilePath* attribute defines the path to the XML file defining this scene. The *defaultSceneName* attribute of *scenes* element contains the name of the scene that is launched as default when the engine is launched and no scene to launch is specified. In our example we have the scene defining file in the folder *scenes* which is contained in the *assets/monolith* folder to keep our files organized.

4.3 Camera

Camera in the scene is represented by a component as well as all the functionality in the engine. By placing the camera in the scene the position and orientation of the camera will be defined by the game object to which it is attached. In case more than one camera is present in the scene the first one will be used. The camera component was already shown in the example of scene defining XML file, its component type is *camera*. In code it is represented by *com.monolith.api.components.Camera* class. *Camera* has three parameters. The *near* and *far* parameters specify the distance of near and far planes from the camera respectively. The near and far planes are the planes parallel with the plane on which content is projected. Everything that is displayed must be between these two planes. The third parameter is *fieldOfView* representing vertical field of view of the camera. The horizontal field of view is calculated from resolution of the surface onto which the content is rendered and this vertical field of view angle.

4.4 Displaying a 3D model

3D model is displayed using *com.monolith.api.components.Model* component. Type of this component for definition in XML is *model*. *Model* has a *meshPath* parameter which contains a path to the rendered mesh or a name of primitive mesh to render. The only supported primitive mesh is *cube*.

4.5 Scripting API basics

In this section I will describe the basic things about scripting. Details about the API introduced in this chapter and also in chapters to follow can be found in the API documentation

on the CD attached to this thesis.

4.5.1 Building a component

As was previously mentioned the scripts are components. To create a component first a subclass of *com.monolith.api.Component* class must be created. Functionality can be added by overriding one or more of the *Component*'s life cycle methods. Following code shows what simplified structure of a component with all life cycle methods looks like.

```
public class Component {
    public void start();
    public void update();
    public void postUpdate();
    public void finish();
}
```

start() Gets called only once at the start of component's life. In this moment component is already attached to its game object. This method should be used to initialize the component.

update() Is used to update the state of game world such as game object's position. Therefore this method is called in every step of game loop.

postUpdate() Gets also called in every step of game loop but is guaranteed to be called after *update()* has been called on all components in the scene. Can be used to do things that need to be done when state of all objects is updated.

finish() Gets called only once at the end of component's life cycle but when component is still attached to its game object. Can be used to do any cleanup or release resources.

Example of working component can be found in appendix B.

Custom component can be attached to any game object in the scene definition XML file by setting a full class name containing the package as component's type, for example *com.example.mygame.PlayerController*. Also any field of this custom component can be set through XML the same way as it is set on built-in components.

4.5.2 Game objects

Game objects are represented by *com.monolith.api.GameObject* class. In order to manipulate any game object first its instance needs to be obtained. Since game objects can be manipulated from components, either the game object to which the component is attached can be retrieved or other game object. In the first case all that has to be done is calling the *getGameObject()* method of component which retrieves the game object to which component is attached. In the second case the game object to which the component is attached

also needs to be retrieved but then the game objects tree needs to be traversed using the *getParent()* and *getchildren()* methods in order to find the specific game object. Helpful thing in this search can be a tag associated with game object. Tag is stored in a public field named *tag* on *GameObject* that can be set to any value by game programmer and then helps with identification of game objects.

4.5.3 Moving a game object

Game objects should be moved from *update()* method of component. Every *GameObject* has a field *transform* which contains a reference to its *com.monolith.api.Transform* which contains its transformation in game world relative to its parent game object and can also provide absolute transformation matrix. To only change the position the *gameObject.transform.setPosition(x, y, z)* needs to be called with *x*, *y* and *z* being the desired position. Similar methods on *Transform* can be used to adjust its rotation or scale.

4.5.4 Communicating with other components

It is often needed to manipulate with other components. To do this the *GameObject* instance offers two ways to retrieve its components. More convenient way is using one of the *getComponent(Class<T>)* or *getComponents(Class<T>, List<T>)* methods. The former takes the class of desired component and retrieves the first instance of this type of component it founds on given game object. The latter also takes a list into which it puts all components of the given type the game object has. The less convenient way is to search for the component yourself in the read-only collection of game object's components which is referenced from *GameObject*'s *components* field.

4.5.5 Accessing the important functionality

Previous chapter mentioned the *Application* object containing all the important functionality. This object can be retrieved by calling the *getApplication()* method on any component.

4.6 Touch input

The *Application* object provides instance of *com.monolith.api.TouchInput* object through its *getTouchInput()* method. The *TouchInput* has a single method *getTouches()* which returns a list of *com.monolith.api.Touch* objects. Every *Touch* object represents one currently running touch gesture, which corresponds to one finger touching the screen. The *Touch* object provides information about the current and the start position of the gesture. *Touch* also contains the state signalizing whether the gesture has just started, is running or ending. This list of touches is valid only for the game loop iteration in which it was retrieved and should be retrieved again in the next one. To track individual *Touch* objects across multiple game loop iterations every object has an identification number which does not change through the existence of the gesture. Example of rotating an object based on touch input can be found in appendix B.

4.7 Collision detection

To use the collision detection, the API contains *com.monolith.api.components.BoxCollider* component which represents a box on the position of its game object of which the collisions are detected. Type of this component in XML files is *boxCollider*. It has multiple parameters to adjust the size of the box, offset of the box from the game object and a parameter *group* determining a group in which the collider belongs. For detailed description of collider groups please refer to section 3.5. To listen for collision events such as collision start and collision end an instance of *CollisionListener* must be registered on the *BoxCollider* component.

4.8 Messaging

Messaging is the mechanism allowing communication with platform on which the engine is running to use the platform dependent services or native UI. In the following sections I will separately describe how messaging is used inside the engine and outside the engine.

4.8.1 Messages inside the engine

Messaging inside the engine is handled using the instance of *com.monolith.api.Messenger*. This object can be retrieved from *Application* object using the *getMessenger()* method. To send the message out of the engine the *sendMessage(Object)* method is used, the only parameter of this method is a message itself which can be any object. To get all messages received during previous frame one of the two methods can be called. First method is *getMessages(Class<T>, List<T>)*. First parameter of the method is class type of the messages you want to receive and the second is a list into which the desired messages will be added. The second method, *getLastMessage(Class<T>)*, is just simplification of the previous one and returns the last received message of the given type in previous game loop step. This method can be used in case only the last received message is relevant, such a case can be when any kind of state is being sent as message and only the newest one is needed.

4.8.2 Messages outside the engine

The instance of *com.monolith.api.external.ExternalMessenger* handles messages outside of the engine. *MonolithFragment* has a public method *getMessenger()* which returns this object. To obtain it from *MonolithActivity* a subclass must be created in which a protected method *getMessenger()* returning the instance of *ExternalMessenger* can be called. Same as the *Messenger* inside the engine the *ExternalMessenger* has a *sendMessage(Object)* method which sends the message inside the engine. The receiving of messages is however quite different. To receive messages from engine a *MessageReceiver* interface must be implemented and registered with the *ExternalMessenger*. *MessageReceiver* is parametrized by the type of messages it receives. Registration is done using the *registerMessageReceiver(Class<T>, MessageReceiver<T>)* taking the class type of received messages and instance of *MessageReceiver*.

4.9 Extending engine's functionality

Extending functionality of the engine can be done by implementing components containing the new functionality and distributing them as standard Java library file with .jar extension. Platform specific functionality must be distributed as a library for the given platform (such as .aar file for Android) and instructions on how to use it must be provided by the developer. One of possible extensions can be for popular application analytics tool Google Analytics. This extension would simply implement one component which would contain methods for sending events. actions and other message types Google Analytics supports. This module would then send the messages out of the engine using the messaging API. Outside the engine simple class would receive the messages and call the appropriate methods from Google Analytics Android SDK.

4.10 Debugging information

Engine supports visualization of colliders. When this option is enabled the colliders are displayed as green wireframe boxes. Boxes turn their color to red when they start colliding and return back to green when collision ends. To turn this feature on a *debug.xml* file needs to be placed into the root of *monolith* folder. Following example shows the contents of this file.

```
<debug drawColliders="true" />
```

The attribute named *drawColliders* sets whether the colliders will be drawn or not. It's default value however is true but this default value is respected only if *debug.xml* file is present and the attribute is not.

Chapter 5

Created applications

5.1 Engine example application

During development of the engine I created a simple application containing examples of different features of the engine and allowing to visually test the engine output. The application contains list with the names of all the examples which is implemented in Android UI framework. While the application contains many of the basic examples in the following sections I will describe the most important ones.

5.1.1 Model importing

This example shows a simple imported model that user can rotate with touch gestures. Figure 5.1 shows screenshot of this example.

5.1.2 Transformation test

Transformation test is more of the visual test of the engine's functions than an example. This test shows a cube that is moved into positive directions of all axes and can be rotated around any of the cardinal axes in their positive directions. The axis of rotation is selected with buttons on the bottom of the screen. Meaning of this test is to verify that the engine's coordinate system is displayed properly and that transformations work properly. By a quick look you can verify that the object is moved into the right direction and that the cube is rotated in the right direction. Transformation test is shown in figure 5.2.

5.1.3 Simple environment

This example, depicted in figure 5.3, shows two main things. First one is that user can manipulate with the world using the Android's UI framework connected to the engine through messaging and the second one is that creation of a simple environment is possible. The scene shows a small environment consisting of multiple scaled cubes. The user can use arrows to fly through this environment and change his orientation with touch gestures.

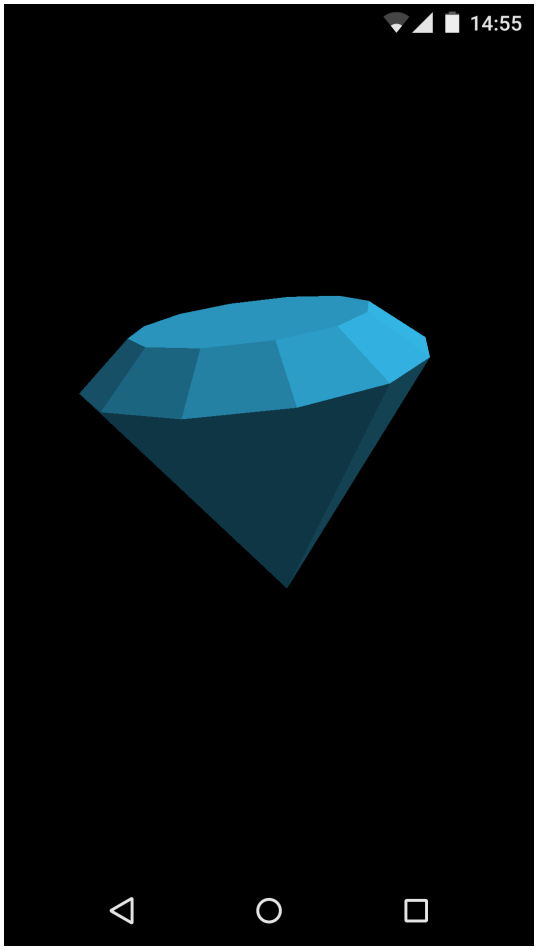


Figure 5.1: The example application screenshot of the imported model

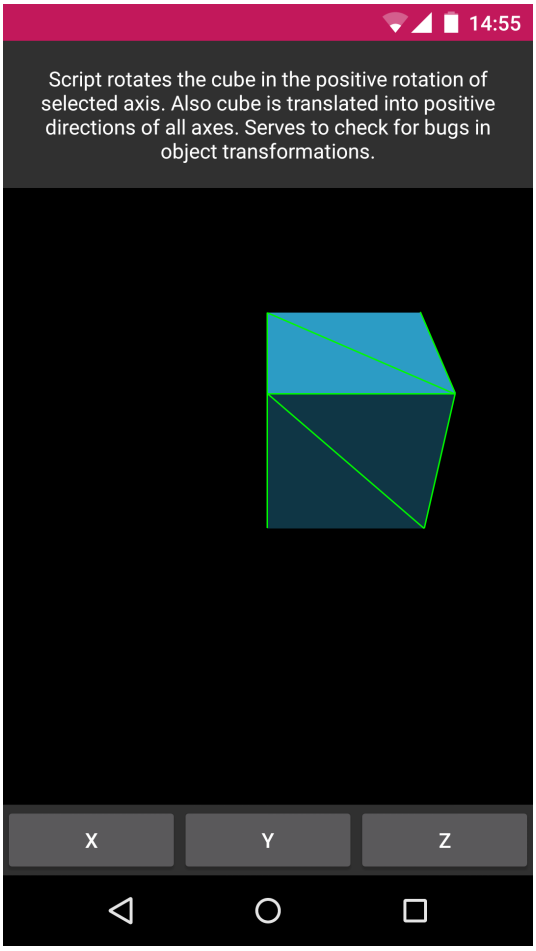


Figure 5.2: The example application screenshot of the transformation test

5.1.4 Performance test

In this example a scene contains 908 objects along the current frames per second (FPS) in the corner of the screen. There are two types of objects, one being the cube and the other being a model of diamond. There is the same number of both objects which are organized into two cones. The example shows approximate performance of the engine, however this test does not perform very well. On LG Nexus 5 the test runs at average 14 FPS. Screenshot of this example is shown in figure 5.4.

5.1.5 Collision detection test

Scene of this example shows 4 cubes with attached colliders. Two of the cubes can be moved and the color of wireframes of cube's colliders shows whether collisions are detected or not. Refer to figures 5.5 and 5.6 for screenshots of this example.

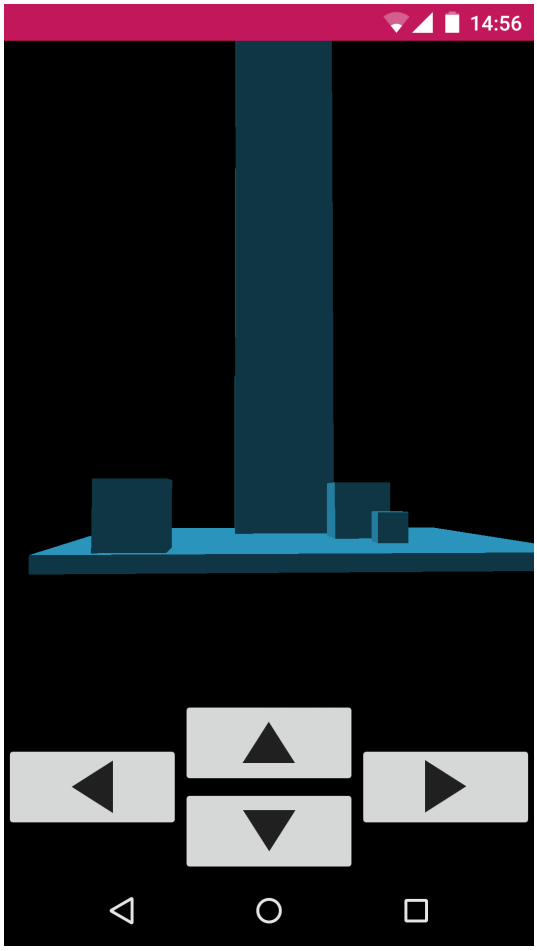


Figure 5.3: The example application screenshot of the simple environment

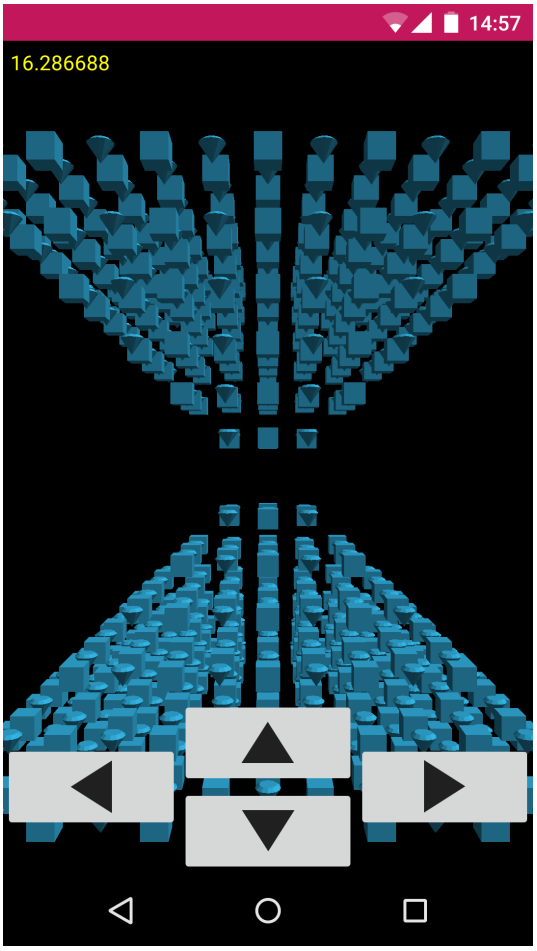


Figure 5.4: The example application screenshot of the performance test

5.2 Showcase game

To fully test capabilities of the engine and how development with the engine feels like I developed a simple endless runner kind of game. Player controls a spaceship that automatically moves forward. The track is randomly generated, possibly infinite and consists of 5 lanes between which player can move. Forward speed is still increasing which makes the game harder and harder. Player must dodge obstacles and can pick up objects that raise his score. Game ends when player hits an obstacle. Player's goal is to reach the best score by picking as many objects and playing as long as possible.

5.2.1 Controls

Game is controlled using the touch gestures. Simple swipe to the left or right moves the player into the next lane in the appropriate direction. Trying to move into next lane when player is on the last lane doesn't end the game.

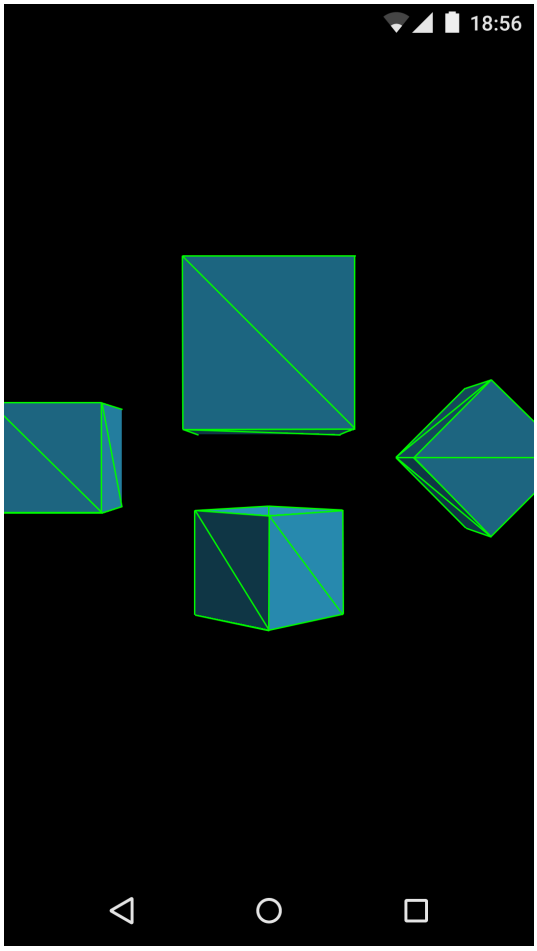


Figure 5.5: The example application screenshot of the collision detection test

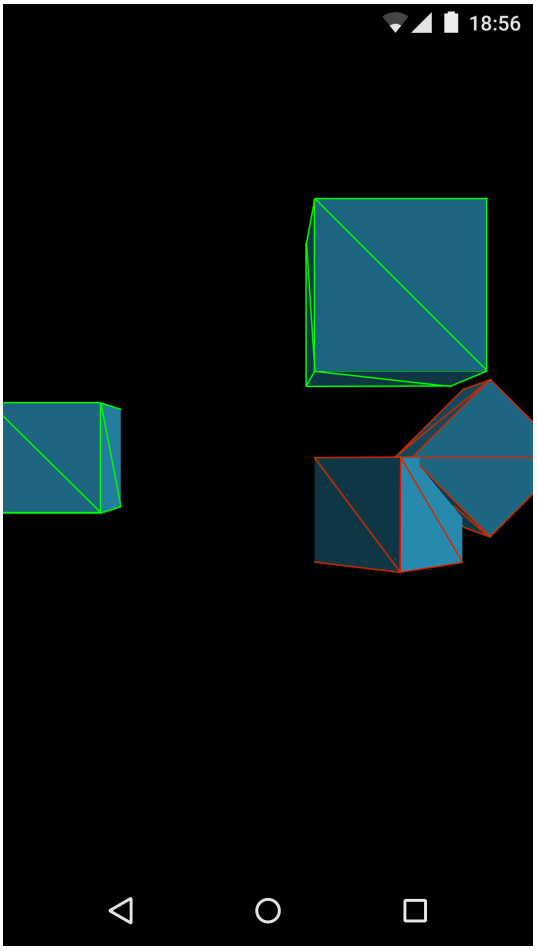


Figure 5.6: The example application screenshot of the collision detection test with colliding objects

5.2.2 Menu

The game has a simple menu as a first screen of the application. The menu allows the user to either start playing or open a high scores screen showing his best scores. Menu and high score keeping is implemented using the Android SDK.

5.2.3 Result

The showcase game was very easy to make which was one of the goals. Code responsible for gameplay consists of 5 components which in total contain 326 lines of code. XML file defining the game scene has 78 lines. The game runs approximately at 60 FPS which is also very good. The game also has so called debug mode which shows colliders as mentioned in section 4.10 and current FPS in the top left corner of the screen. Screenshots of the showcase game can be seen in figures 5.7 and 5.8 while figures 5.9 and 5.10 show the game in debug mode.

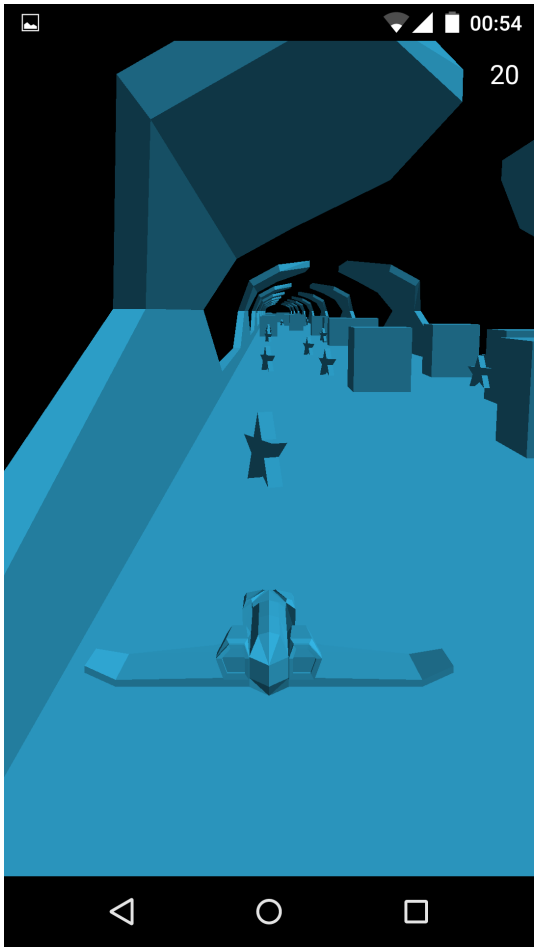


Figure 5.7: Screenshot from the showcase game

Figure 5.8: Screenshot from the showcase game

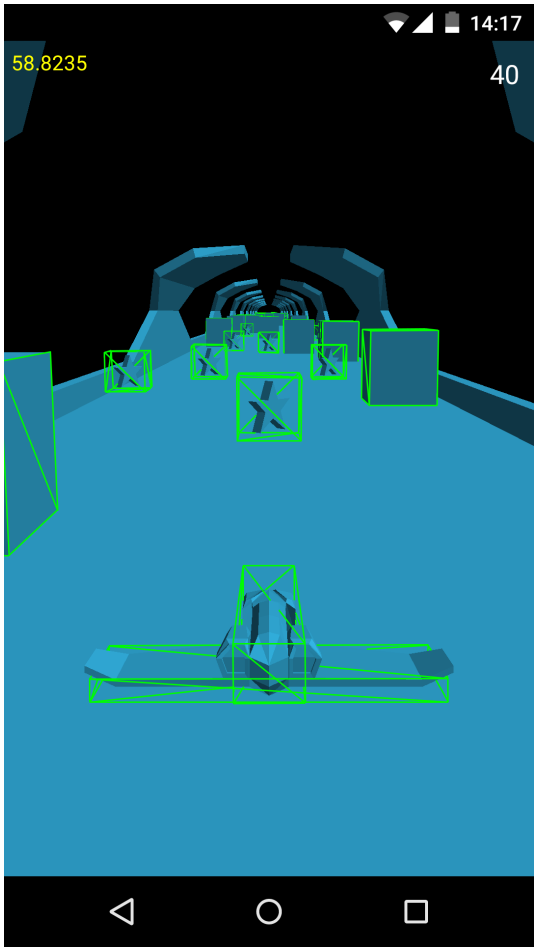


Figure 5.9: Screenshot from the showcase game in debug mode

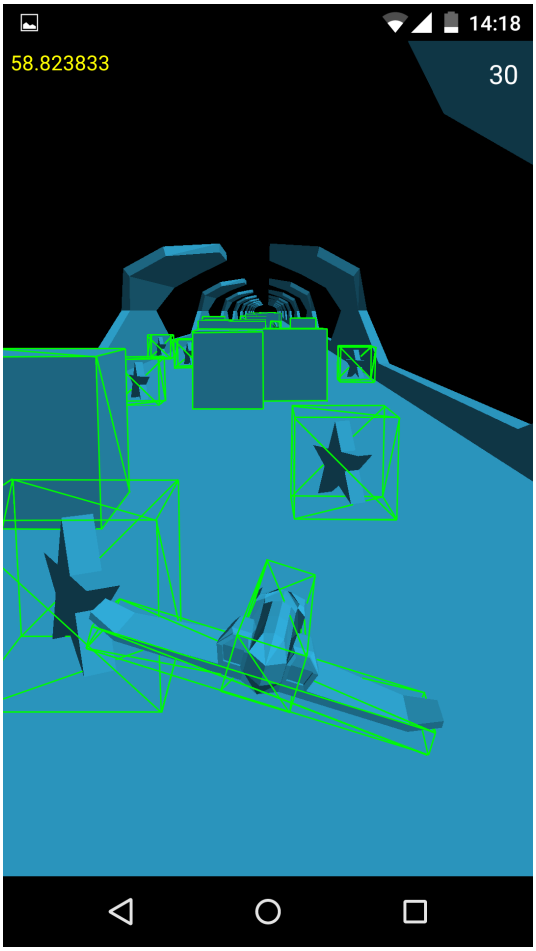


Figure 5.10: Screenshot from the showcase game in debug mode

Chapter 6

Testing

During the development appeared the need to test such things as right order of calling the component's lifecycle methods and if they are called at all and various other concepts. For this reason unit testing has been enabled and a few tests written. Testing uses the jUnit testing framework[1]. To be able to test the engine the *Engine* instance as described in section 3.6 is initialized with dummy objects. Whole engine can be launched as usual in unit test and anything can be tested. For every test there must be created a scene that is launched by the engine. To do the testing itself components must be implemented to assert the tested concepts or simulate behavior.

6.1 Component life cycle test

The correctness of the order of the calls to the component's methods and ensuring that all methods are called is crucial. To test this the test uses a component that asserts that all methods are called and calls are done in proper order. The scene contains three objects where two of them are children of the third object and the two are siblings. The complete source code of the life cycle testing component is attached in the appendix C.

6.2 Game object tree manipulation test

The tree of game objects can be manipulated in any way from the code of components. Therefore it is important to ensure that the life cycle of components that are either manipulated or attached to manipulated objects is still valid. To do this the test uses the component asserting the life cycle correctness from previous test and manipulates with game objects that have instances of this component attached.

6.3 Messaging test

To test messaging a simple component that upon receiving one message sends out other message was implemented. The test simply sends a message and asserts that the answer

[1] <http://junit.org/> (May 18, 2015)

comes. The testing scene contains a single game object with single instance of this component.

6.4 Drawbacks of testing

Unfortunately the structure of tests is not very handy. The testing scenes are separated from the code that is running in them. Also to create a single test at least one scene XML file and usually two additional Java files must be created. One for the initialization of the test itself and the other for component asserting or simulating the behavior. This causes a lot of repetitive work and makes creation of tests unpleasant.

Chapter 7

Future work

In its current state the engine is more of a prototype than a usable product. A lot of work is still needed to reach the point where the engine can be used for development of real games. In the following chapters I will describe some of the major changes that are needed or recommended to reach this state as well as some minor concepts that can be added and some possible optimizations.

7.1 Performance optimization

The performance test in the example application shows that when there is a lot of objects in the scene the FPS significantly drops. Therefore one of the first tasks on the list should be optimizing the performance.

7.2 Graphics

So far development was focused on architecture and scripting API instead of graphics features so this is one of the important areas that would need an improvement. Most important of the graphics features should definitely be the support of different types of lights and the ability to move these lights. Next in line is texturing along with skybox support. Last but not least is the support for creation of custom surfaces of objects. One of the ways to implement this can be to allow users to create their own shaders which gives game programmers high degree of freedom.

7.3 Systems

Systems are a simple concept that should allow better implementation of things that are supposed to be present only once in the game scene and are not meant to be positioned anywhere in the scene but only run in background. Such things include collision detection system that only searches for collisions between colliders but is not placed anywhere in the scene. In the current implementation to create something like this user must create a component and add it to object in the scene. With systems this unnecessary game object would be removed.

Systems would work very similar to components. They would also feature life cycle methods to do initialization, update and reaction to updated state of game world. However unlike components they would not be attached to game object but to scene itself.

7.4 Collision detection

As mentioned in section 3.5 the collisions are tested between every pair of colliders in the scene. This can be optimized by using the collider groups but this optimization takes time to do and will not be sufficient in more complex scenes with more moving and colliding objects. Therefore the first optimization on schedule is using a bounding volume hierarchy to allow omitting of collision testing between obviously not colliding objects. Ericson (2004) [2] and Bergen (2003) [7] provide more information about bounding volume hierarchies.

Collision detection can also be improved in terms of collider types. Box will not be good approximation of an object in most cases and so more types should be added such as a capsule and eventually system should detect collisions between the 3D models or their low polygon count approximation. Also the information about the colliding triangles should be available from collision detection framework. Eventually raycasting should be added since it is an essential feature for whole genre of shooter games.

7.5 Extending the engine to other platforms

With current implementation it is possible to very easily extend the engine to platforms supporting Java. With LWJGL library[1] it is possible to extend the engine to Windows, OS X and Linux. However, to extend the engine to such platforms as iOS the completely rewritten version of the engine in C++ language would be needed. Implementation in C++ would also improve performance and is therefore one of the desired paths to take.

[1] <http://www.lwjgl.org/> (May 18, 2015)

Chapter 8

Conclusion

Thesis introduced design of a game engine and described how implemented prototype is used. Chapter 2 listed important characteristics of game engines, all of which were projected into the engine without any problems. Usability and fast prototyping possibilities were proven by implementation of showcase game. Engine allows genre independence since every feature of the engine is very general such as placement of the camera in the scene. It is ready to be expanded since developing and distributing new features is very easy. Communication with underlying platform was fully implemented and allows very easy connection from the engine. Project is very nicely structured and platform independent code clearly separated from Android code which allows immediate expansion to platforms running Java. In future multiplatform engine can nicely use version of the platform independent code rewritten into such language as C or C++.

Performance of the engine is enough for the showcase game which runs approximately at 60 FPS all the time on LG Nexus 5. However, in the performance test described in section 5.1.4 the FPS stays at 15 on the same device which is very low since the standard is 60 FPS for fast action games like this one. Engine definitely needs performance tweaks and should be able to handle the performance test running on 60 FPS on the mentioned device.

The project brought me a lot of knowledge and experience with implementation of games and game engines. The implemented prototype can also serve for further experimentation with other concepts and features. From this point of view for me the project was definitely a contribution. The implementation of showcase game showed that the engine is usable. Process of game creation is pleasant without requiring any unnecessary work. The API is simple to use and whole engine allows quick implementation of games.

In conclusion the result is very small game engine which is ready to be expanded and built upon. It also provides important features and properties needed for succeeding in modern game development world.

Bibliography

[1] David H. Eberly. *3D Game Engine Design : A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann, 2000.

[2] Christer Ericson. *Real-Time Collision Detection*. CRC Press, 2004.

[3] Ian Parberry Fletcher Dunn. *3D Math Primer for Graphics and Game Development*. A K Peters, CRC Press, 2nd edition, 2011.

[4] David Graham Mike McShaffry. *Game Coding Complete*. Course Technology Cengage Learning, 4th edition, 2013.

[5] Jesse Schell. *The Art of Game Design: A Book of Lenses*. CRC Press, 2008.

[6] Martin Štýs. Komponentový přístup při tvorbě objektů herní knihovny. Master's thesis, ČVUT FEL, 2014.

[7] Gino van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, 2003.

List of Figures

Appendix A

Nomenclature

3D	three dimensional
FPS	frames per second
GPU	graphics processing unit
OBB	oriented bounding box
UI	user interface
VBO	vertex buffer objects

Appendix B

Component example

Following example component rotates game object to which it is attached according to touch input gestures.

```
package com.monolith.showcase.engine;

import com.monolith.api.Component;
import com.monolith.api.Touch;

import java.util.List;

/**
 * This is simple rotation controlling {@link
    com.monolith.api.Component}.
 * It rotates it's {@link com.monolith.api.GameObject} according
    to touch input.
 */
public class TouchRotationController extends Component {

    private static final float FACTOR = 0.1f;

    private int mLastTouchId = -1;
    private float mLastTouchX;
    private float mLastTouchY;
```

```
    @Override
    public void update() {
        List<Touch> touches =
           getApplication().getTouchInput().getTouches();
        if (touches.size() > 0) {
            Touch touch = null;
            if(mLastTouchId != -1) {
                // Try to retrieve the Touch we tracked last time
                for(int i = 0; i < touches.size(); ++i) {
                    Touch ithTouch = touches.get(i);
                    if(ithTouch.getId() == mLastTouchId) {
                        touch = ithTouch;
                        break;
                    }
                }
            }
            if(touch == null) {
                touch = touches.get(0);
            }

            float currentTouchX = touch.getX();
            float currentTouchY = touch.getY();

            if (touch.getState() != Touch.STATE_BEGAN &&
               touch.getId() == mLastTouchId) {
                // Factor needs to be scaled according to screen
                   pixel density
                // since touch coordinates are in screen pixels
                float countedFactor = FACTOR /
                   getApplication().getDisplay().densityScaleFactor;
                getGameObject().transform.rotateBy(
                        -(currentTouchY - mLastTouchY) *
                           countedFactor,
                        -(currentTouchX - mLastTouchX) *
                           countedFactor,
                        0);
            }
            if(touch.getState() == Touch.STATE_ENDED) {
                mLastTouchId = -1;
            } else {
                mLastTouchId = touch.getId();
            }
            mLastTouchX = currentTouchX;
            mLastTouchY = currentTouchY;
        }
    }
}
```

Appendix C

Component life cycle testing component

```
package com.monolith.tests.component_lifecycle;

import com.monolith.api.Component;

import java.util.ArrayList;
import java.util.List;

import static org.junit.Assert.*;

/**
 * Component asserting correctness of it's own lifecycle.
 */
public class LifecycleAssertingComponent extends Component {

    private static List<LifecycleAssertingComponent> sObjectCache
       = new ArrayList<>();

    private boolean mStartCalled = false;
    private boolean mUpdateCalled = false;
    private boolean mFinishCalled = false;

    private String mTag;

    public LifecycleAssertingComponent(String tag) {
        this();
        mTag = tag;
    }
```

```
public LifecycleAssertingComponent() {
    super();
    sObjectCache.add(this);
}

@Override
public void start() {
    assertFalse(getMessage("Start is being called for second
       time"), mStartCalled);
    mStartCalled = true;
}

@Override
public void update() {
    assertTrue(getMessage("Start was not called before
       update"), mStartCalled);

    assertFalse(getMessage("Update is being called for second
       time"), mUpdateCalled);
    mUpdateCalled = true;
}

@Override
public void postUpdate() {
    assertTrue(getMessage("Start was not called before
       postUpdate"), mStartCalled);

    assertTrue(getMessage("Update was not called before
       PostUpdate"), mUpdateCalled);
    mUpdateCalled = false;
}

@Override
public void finish() {
    assertTrue(getMessage("Start was not called before
       finish"), mStartCalled);

    assertFalse(getMessage("Finish is being called for second
       time"), mFinishCalled);
    mFinishCalled = true;
}
```

```
    /**
     * This method should be called when this component is removed
     * from it's GameObject or if Engine's life ended.
     */
    public void checkEverythingOkInTheEnd() {
        assertTrue(getMessage("Start was not called during this
            component's lifecycle"), mStartCalled);
        assertTrue(getMessage("Finish was not called during this
            component's lifecycle"), mFinishCalled);
    }

    private String getMessage(String message) {
        if(mTag == null || mTag.length() == 0) {
            return message;
        } else {
            return message + " for component " + mTag;
        }
    }

    /**
     * This method should be called when Engine's life ended to
        ensure
     * all component's of this class had proper lifecycle.
     */
    public static void checkObjectsInCacheAreOk() {
        for(LifecycleAssertingComponent component : sObjectCache)
            {
            component.checkEverythingOkInTheEnd();
        }
    }

    public static void clearObjectCache() {
        sObjectCache.clear();
    }
}
```

Appendix D

Contents of the CD

Following diagram show the structure of files on the attached CD.

```
/
├── thesis
│   ├── source ..... Source for the thesis
│   └── thesis.pdf ..... Thesis pdf file
├── sources
│   ├── engine ..... Sources for the engine with example application
│   └── game ..... Sources for showcase game
├── doc ..... Javadoc documentation of engine's API
├── lib
│   ├── monolith_core_v0.3.jar ..... Core part of the engine library
│   └── monolith_android_v0.3.aar ..... Android part of the engine library
└── exe
    ├── example.apk ..... Example application
    ├── showcase-game.apk ..... Showcase game
    └── showcase-game-debug.apk ..... Showcase game in debug mode
```