Hacettepe University

Department of Computer Engineering

BBM429 – Project I

Hoppy

2D Game & Engine Development for Android

Furkan KÜÇÜKDEMİR

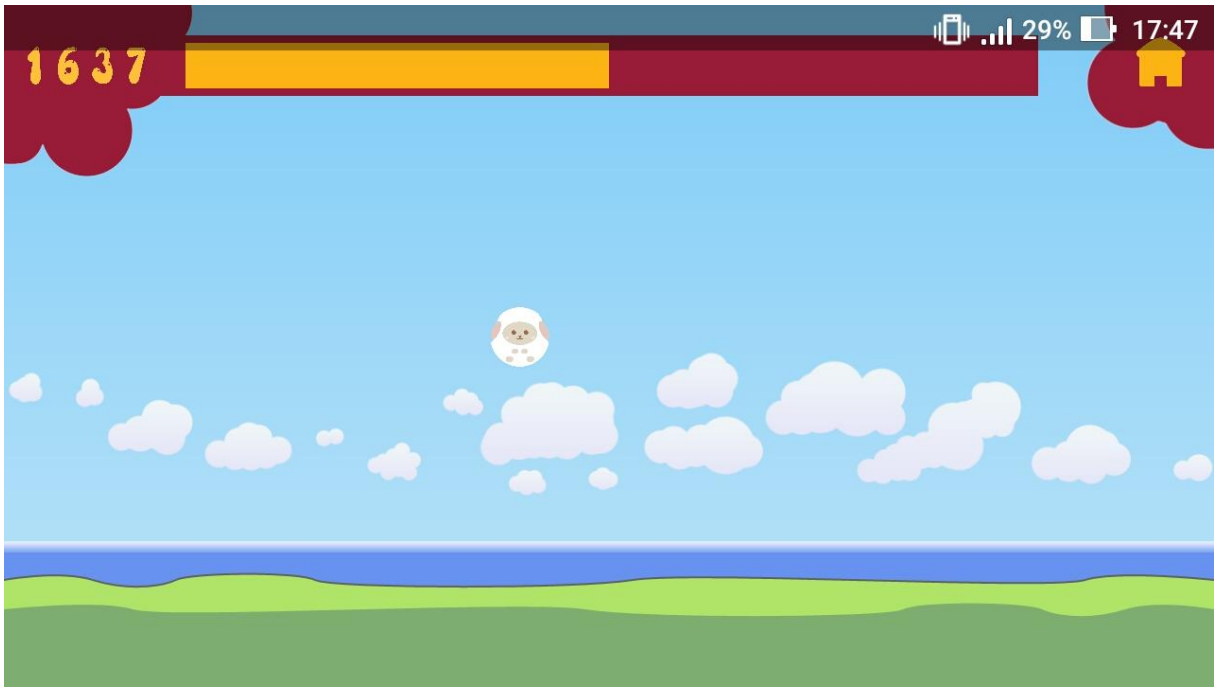
21328213

b21328213@cs.hacettepe.edu.tr

OVERVIEW

Hoppy is a 2D platformer game project for Android operating system. The game is developed using a custom engine which is also developed during the project. The game engine is temporarily named “MEngine”.

The final screenshot of the game can be seen below :



TOOLS USED

The list of tools used during the project is as follows :

- Android Software Development Kit
- Android Native Development Kit
- Git
- Vim

In addition to the tools above, Ubuntu 14.10 is used as operating system during the development.

Android Software Development Kit

Android Software Developement Kit(SDK) developed by Google is a bundle of tools which are needed in order to develop applications on Android platform. It includes tools for building applications, documentation on various topics, example projects and tools for debugging and profiling.

In the project, tools below are used :

- adb : Android Debug Bridge is a tool used for communicating with the development machine. It is mainly used to install the application on the device.

- monitor : Android Device Monitor is a tool used for application analysis and debugging. It provides a way to monitor log messages. It also can be used to explore files on the device.
- aapt : Android Asset Packaging Tool is used to create, update and view application packages. It compresses resource files, compiled Java bytecodes and compiled machine codes into archive files.
- apkbuilder : Android Application Package Builder is used to build apk files.
- dx : Dx is used to convert compiled Java bytecodes into a dex file.

Android Native Development Kit

Android Native Development Kit (NDK) is a set of tools offered by Google in order to develop applications for Android in native programming languages like C/C++. Even if the majority of applications for Android are implemented in Java, applications which use CPU heavily like game engines, physics simulations and signal processing applications are written in C/C++ in order to gain performance improvements. In addition to that, NDK makes it possible to use libraries which are already written. So, developers can reuse their code from previous projects or other projects.

Android NDK provides an API for developers in order to access physical components of the device like sensors and touch screen. In addition to that, NDK includes a set of tools to compile C/C++ codes for different architectures (i.e. ARM, MIPS, x86) and debug them. Also, there are example projects which demonstrate the usage of NDK.

Git

Git is an open-source version control system tool. It is used to improve speed and efficiency of the developers during the development of a project. It is originally written by Linus Torvalds in 2005.

During the development of Hoppy, it is used to track the progress of the project and mark milestones.

Vim

Vim is a text editor for Unix systems. It is really lightweight and easy to use. It is highly configurable and supports a large number of plugins.

DEVELOPMENT

The very first step of the project was preparing the development environment. Since I have used Android Studio and Eclipse before and have found them slow, I chose not to use them. Instead, a build script is written using tools in Android SDK and Android NDK. Then, git is initialised on the project.

After the development environment is prepared, the boilerplate code of Android Native App Glue is written. Using Android Native App Glue developers can write fully native (i.e. C/C++) applications for Android. So, it was a perfect option for the project.

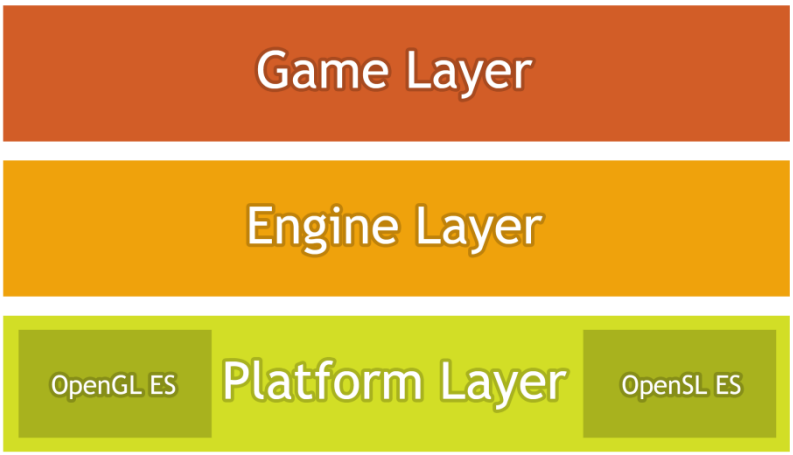


Figure 1 – Architecture of the engine

The project is structured as shown in Figure 1.

The lowest layer is platform layer and platform specific problems like memory allocation, file input/output, application's life cycle, inputs are handled there. Also, the initialisation of the graphics API which is OpenGL ES for Android platform is handled in the platform layer.

At the engine layer, problems which are not different on different platforms are solved. For instance, collision detection, entity-component system and asset management are handled in engine layer.

Game layer is used to handle game-specific problems such as what will happen if character A hits Enemy X or how fast should the character move.

The implementation is done starting from the lowest layer. After the boilerplate code of Native App Glue, OpenGL ES is initialised. Since it is officially supported by Android, it was not so difficult to implement. The tricky part was determining how to move the data that is required for rendering graphics between the game layer and the platform layer. In order to make it flexible and reusable across different graphics APIs, a structure which is called "render command" is used. By using render commands, the engine is able take rendering commands and at the end of the frame it interprets these commands and takes appropriate action according to platform's graphics API.

After that, timers are introduced and the game loop is constructed. In each iteration, it updates the game world according to user input and then renders the current state of the game world. While doing that, it keeps the track of time and if the time spent for a frame is longer than 16 milliseconds (which is 1/60, where 60 is target frames per second) it accumulates the difference. If that accumulation becomes greater than 16 milliseconds at some point, the game world is updated more than one time.

In pseudocode, we can represent the game loop as follow :

```
While Game is Running
      Add the time difference to accumulation
      While accumulation is greater than 16ms
            Update the game world
      Render the game world
```

At this point, the engine was capable of render game objects at a fixed interval. The next step was handling user inputs. To do that, Input API provided by Android NDK is used. Using that API, the engine is able to sense key and motion (i.e. touch) events. With the help of documentation and descriptions in header files, it was not so difficult to integrate it with the engine.

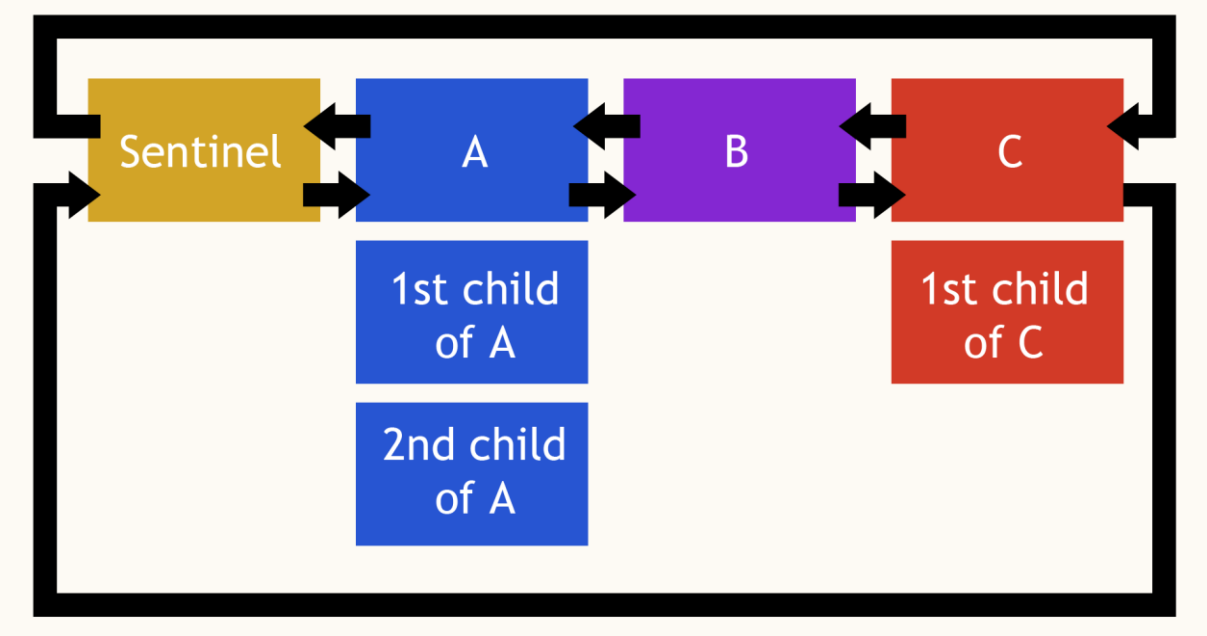


Figure 2 – Structure of the list of entities and compoents

The next thing to do was implementing an entity-component system. In order to solve this problem, a sentinel is created at the initialisation stage of the engine. That sentinel is used as a reference point for the entity linked-list. Then, whenever an entity is created in the game, it is added to that list. Each entity holds three pointers : A pointer to next entity, a pointer to previous entity and a pointer to a subentity. So, it is possible to create entities hierarchically. The structure of the list can be seen in Figure 2.

In order to make entities more powerful, components are implemented. Components are used to give some specific attributes to entities. For example, if an entity is supposed to move in the scene, it has to have a rigidbody component. The list structure of the entites are applied for components and a component sentinel is added into the structure of the entity.

While testing the entity-component system, a bug which is caused by render commands was found. Render commands were being generated during the game world update and if the game was updated more than once due to time accumulation, render commands were also being generated more than once and that was causing the bug. An object was appearing at

two different positions on the screen which makes the game look laggy and visually unpleasant. In order to fix that bug, generation of render commands are seperated from the game world update. In this new scheme, the whole list of entities are traversed after the game update and proper render commands are generated during that traversal for each entity. So, the game loop has become as follows :

```
While Game is Running
      Add the time difference to accumulation
      While accumulation is greater than 16ms
            Update the game world
      Generate render commands
      Render the game world
```

While implementing that seperate pass, another bug was encountered and it was a bit difficult to locate the root of that bug. As a consequence, a very simple call stack is implemented. In order to make it work, a marker (BeginStackTraceBlock) has to be put at the beginning and another marker (EndStackTraceBlock) has to be put at the end of all functions in the project. Even if it was lousy, it helped a lot to solve various problems during the development.

After that, the next part of the engine to implement was asset management. There are many types of assets like bitmaps, fonts, sound effects etc. Currently, the engine supports only bitmap assets. But the asset manager is structured so that it is easy to support new asset types. A tool for packing assets is written first. This tool takes assets (e.g. bitmaps) as input and outputs a formatted asset file. This file's format is as follows :

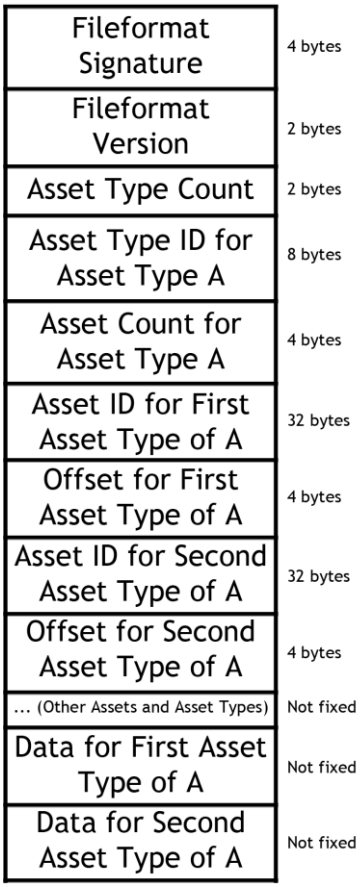


Figure 3 – Structure of Asset File

The asset packer's output is copied into the project folder. Then, it is read by the engine at the initialisation stage and whenever an asset is needed during gameplay, asset manager supplies desired asset. Currently, whole asset file is read into the main memory by the engine. However, it is not feasible for bigger games.

Completing asset management made a big difference visually. Thereafter it was time to implement the gameplay. The first thing implemented for that purpose was proper movement of game objects. When the input handling was done, a simple movement function was implemented but it did not look good. According to that simple scheme, if an input was detected, the character was moving towards left or right with a constant velocity. The new version of the movement is a bit more realistic and it updates the position of a game object according to its velocity and acceleration. For example, whenever the user touches to the screen of the device, the character accelerates according to the input.

At that point the character was moving around the game world and the user was able to control it. But there were not any enemies or any other game objects. Since the game is planned to be using a fixed camera(i.e. non-scrollable), the character will not encounter with enemies during the gameplay. So, enemies had to come into the scene and enemy spawners are implemented for that purpose. There are two spawners in the scene and one of them is located at the left edge of the screen and the other one is located at the right edge of the screen. They generate enemies at some interval and that interval can be a fixed value or a random value selected between a specified minimum value and maximum values. Also, they spawn enemies at different position and with different velocities to avoid a monotonous gameplay.

The game started to feel like a game as soon as spawners are implemented. Nevertheless there were still no interaction between game objects. With that in mind, collision detection is implemented. There are two types of colliders supported in the engine : circular collider and rectangular collider. As a result there can be three types of collisions between those colliders : collisions that occur between two circular colliders, collisions that occur between two rectangular colliders and collisions that occur between a circular collider and a rectangular collider. Each of those collisions are handled using Minkowski sum of colliding shapes.

- Circle-Circle Collisions : Assume that the radii of the circles' are R1 and R2. Then, a new circle is created with a radius (R1+R2) and positioned at the location of the first circle, P1. If the center of the second circle, P2, is inside that new circle then there is a collision between two circles. Otherwise they do not collide.

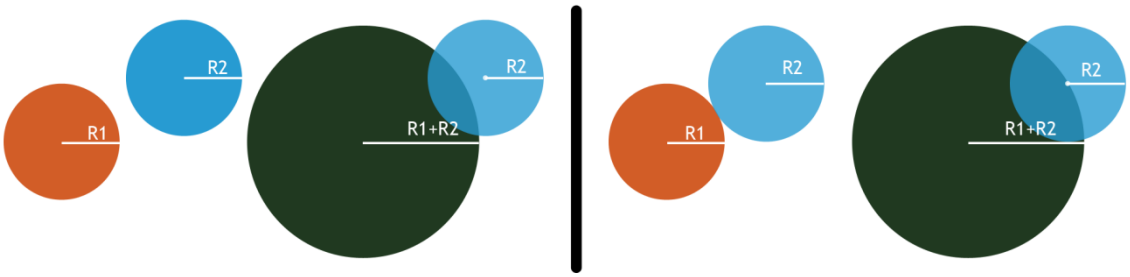


Figure 4 – Collision Test Between Two Circular Colliders

- Rect-Rect Collisions : Assume that the size of rectangles are W1 x H1 and W2 x H2. Then, a new rectangle is created which has the size (W1+W2) x (H1+H2) and it is positioned on P1 which is the position of the first rectangle. If the center of second rectangle, P2, resides in that new rectangle, then there is a collision between them. Otherwise, they do not collide.

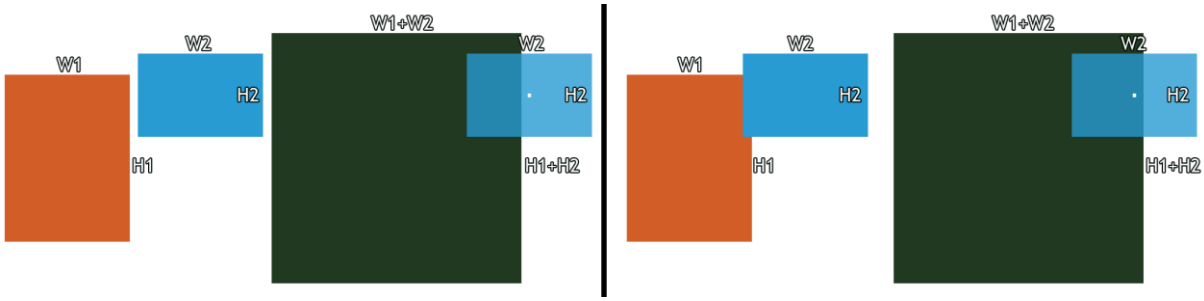


Figure 5 - Collision Test Between Two Rectangular Colliders

- Circle-Rect Collisions : Assume that the size of the rectangle is WxH and the radius of the circle is R. By using these two shapes, a new rounded rectangle is created as shown in Figure 6 and positioned at the center of the rectangle. If the center of the circle is inside the rounded rectangle then there is a collision. Otherwise, the rectangle and the circle do not collide.

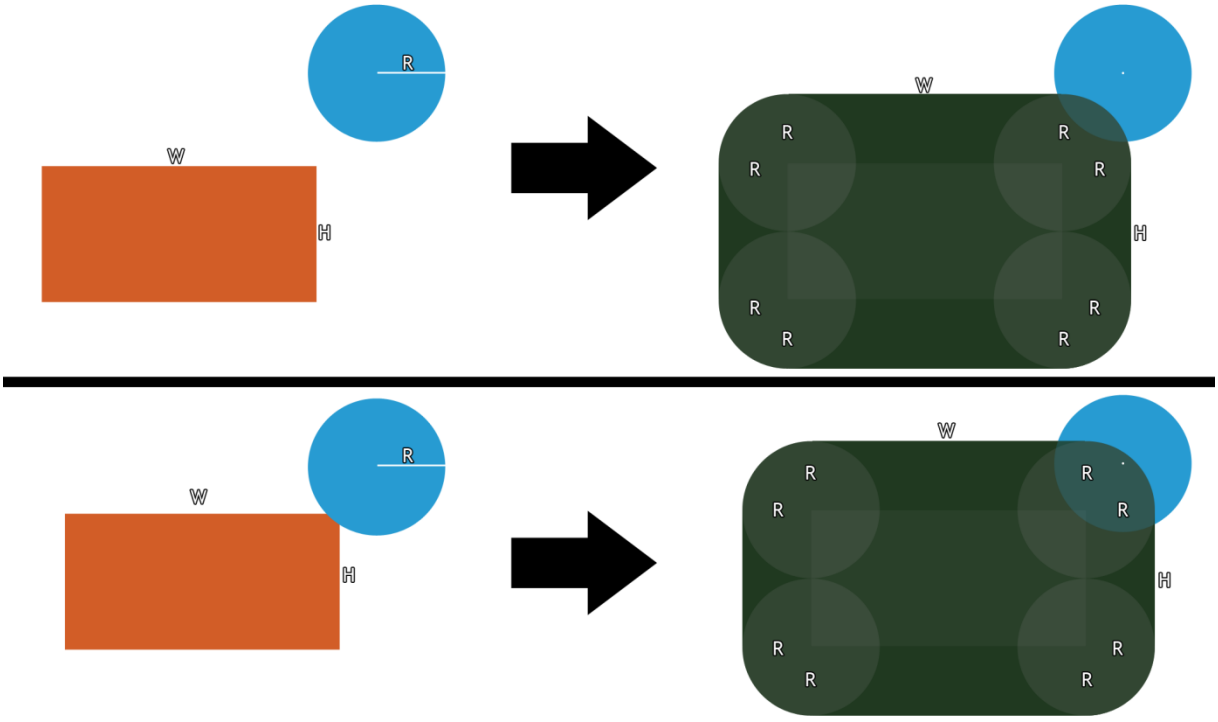


Figure 6 - Collision Test Between A Rectangular Collider and A Circular Collider

If a collision is detected between two game objects in the scene and if those objects have a rigidbody component which provides the ability to move in the game world then the velocity of those objects are changed according to the objects' contact point. But changing the velocities with realistic physics rules did not go well hand to hand with the planned gameplay rules. Characters and enemies are obliged to have a constant jump height and it

must not change during a game session without an intention (e.g. by using powerups). As a result, velocity changes are arranged in accordance with the gameplay rules. There are colliders on the character, enemies and the grass in the game world. In the future, for example, if some powerups are added to the game world, they should also use colliders. But instead of affecting the character's movement direction, they will add or change some properties of the character. For instance, one of them may refill the character's health.

After implementing the collision detection and completing the movement rules of the game objects, health and attack components are added to the game. Health component is created for the character and attack component is created for the enemies. Whenever a collision occurs between the character and an enemy, the health of the character is decreased according to the enemy's hit point. Even if there is just a single type of enemy in the game world currently, it is easy to add new enemies with different hit point and jump height values.

So by the implementation of those components, the gameplay part of the game was completed. But the gameplay was immediately starting when the user ran the application. In other words, there were no menu and screen transitions. Also, there were not any actions specified when the health goes down below 0. As a consequence, a main menu with a play button and an end screen which shows the score are implemented. The end screen also contains a replay button and a menu button.

Besides from these, the lack of an user interface was making the game look really poor. So, a scoreboard, a health bar and a menu button are added to the game screen. The problem was that the engine was not supporting font assets and font rendering. Proper handling of fonts is not a trivial task. It requires to get familiar with TrueType format and involves a bit complex things like kerning, spacing etc. Due to time constraints fonts are not handled as they should be in the project. Instead, 256 by 256 pixels sized bitmaps of the characters are used for them. Since fonts are required only to write score of the player, only numeric characters are handled in the project. This should be fixed as soon as possible since this is not a reasonable solution for a game engine.

The last thing done in the project is playing a background music. For this purpose, an audio manager is created. OpenSL ES is used in order to control that audio manager. OpenSL ES is a convenient choice to communicate with the audio hardware and it was easy to embed it in the game engine. Currently, the game engine does not support sound effects since Hoppy does not include them. The only audible content in the game is the background music.

After the implementation of the audio manager, the project has reached the planned status and the game was ready to play.

KNOWN ISSUES

Even if the game is playable and works without an error on the test device, it is not tested on any other device. Some problems may occur on devices with different screen resolutions and different application binary interfaces(ABIs). Also, there are some other issues known and has to be fixed. The list of those are as follows :

- Application's state changes are not handled properly. For example, if a phone call is received in the middle of the game it may cause problems.
- Device's screen dimensions and screen refresh rate are not detected properly. Functions on the native side were returning false information about those and they are removed from the project. It requires a call to Java side using JNI but it is not done yet.
- Layer abstractions got blurred towards the end of the project and some portions need to be refactored. For example, game code should be completely seperated from the engine code so that it should be loaded as a dynamic library.
- Software renderer should be completed and support for different graphics APIs should be added. This will help to abstract graphics layer properly.
- OpenGL ES implementation should be revisited. Some techniques can be applied in order to reduce state changes and improve performance. For example, texture atlases can be used or render commands can be sorted by their shaders and textures.
- The game should be tested on different devices. There are some parts in the engine that behave differently according to the device's ABI. For example, timer is implemented in this way. It is implemented for different ABIs but not tested and not guaranteed to work. Also, shaders should be managed better.
- Asset management should be improved. An asset cache can be created and if the desired asset cannot be found in the cache, it should be loaded from the disk dynamically.
- Fonts should be handled properly. Currently neither their rendering nor their loading as asset handled as they should be.
- User inputs and user interface elements can be handled faster. Not all of the entities are interested with the inputs and input handling is wasting time traversing the whole list of entities in the current version of the game engine. Input handling can be done for both game objects(i.e. entities) and user interface elements in a single function so that it will be faster.

CONCLUSION

Developing a 2D game engine helped me to gain experience and to become aware of the problems to be solved. The hardest part of the project was keeping the architecture stable all the time. Due to inexperience and time constraints, the abstraction of different layers cannot be done as desired and it still requires some work. Developing a different game for a different platform using the project's codebase can help for that purpose. By doing so, platform-specific parts and game-specific can be seen clearly.

REFERENCES

https://en.wikipedia.org/wiki/Android_software_development

https://developer.android.com/studio/command-line/index.html

https://developer.android.com/ndk/index.html

https://git-scm.com/about

http://www.vim.org/

https://www3.cs.stonybrook.edu/~algorith/files/minkowski-sum.shtml