# Run Ninja, RUN!

Jamie Haddow
CMP208
0705082

## Contents

## 1.0 Introduction

The purpose of this created application, is provide evidence of progress for the University of Abertay's second year module CMP208 – “Game Programming and System Architectures”. The function of this application, is to showcase a game, working on Sony's PlayStation Vita hardware and supplied GEF framework(With modifications taken into account due to the coronavirus restricting students access to the hardware which will be explained in 4.0 User Guide).

The main features of this project are:

- Heavy model usage, with 44 models being used in total.
- A strong use of Enumerators for menu/game choices.
- Numerous music and audio tracks,
- Physics and collision's using the Box2D framework.
- Animation on the player model(which was not part of the course content)
- Keyboard input for menu and player interaction
- A simple scoring system

## 2.0 Application Design

This application consists of six user made .cpp/.h classes(That were not provided during lab work). After considering the use of an object manager for the roads,buildings and obstacles, it was decided that a single class for each would be more appropriate. Taking the roads as an example, due to the 15 road models all working together in the sense of 'one object' and not acting independently, creating an array(the amount of road models needed was already known, so an array was chosen over the use of a vector) of road class objects would be an over complication of code. The building and obstacle classes also fall into this category/consideration.

Each class(minus the collision.h), has a single Render and Update function, that are then called in the main .cpp “Scene_app” that handles the main game logic. There is also another .cpp titled “ Zombies” which handles the animated models for the end scoring screen. This was not suitable to be placed inside any of the other already created classes so its own .cpp was created. For collisions, instead of inheriting from the Gameobject class which also inherited from Meshobject, a simpler class was create called “Collisions”. Player and Obstacles then inherited from this.

In this application, there are three “areas” the player will traverse until the the end goal is reached. Military themed, Samurai themed and Apocalypse/zombie themed. Due to knowing the length of each “area”, a specific distance variable was created, and then added onto the next road model in the array of mesh objects. The buildings were also handled the same, except slightly off the origin on the Z axis to remove the first building in each area appearing on the origin of each area.

```
create an integer distancevariable = 0;
create a for loop for the number of mesh objects
{
create a random number 0-4
set the mesh for the mesh object using the random number above to correspond with what road model to use
create a vector using the  distancevariable  as the x value
translate the model using a matrix
increment the distance variable, distancevariable = distancevariable +20(this being the width of the road model)
}
```

For this reason, an object manager was not an appropriate class structure for this game. On further reflection after learning more about Data oriented Design, this would be changed which is explained further in the **5.0 Data Oriented Design**

## 3.0 Techniques Used

Due to this application using a large number of models, it was required to do research into the understanding and use of 3d modelling and texturing in Maya. There were issues using the provided model loader to convert fbx files in scn files. For example, the player had to be scaled down to a scale of 0.01 on x,y,z axis's. This required a number of extra vectors and matrices to work correctly. The student found that converting an fbx file to an obj, and then saving that obj to  an fbx took any model scaling,rotation and transforms into consideration. This specific instance was left in place for an example of using a scaling matrix, but future models were all scaled properly.

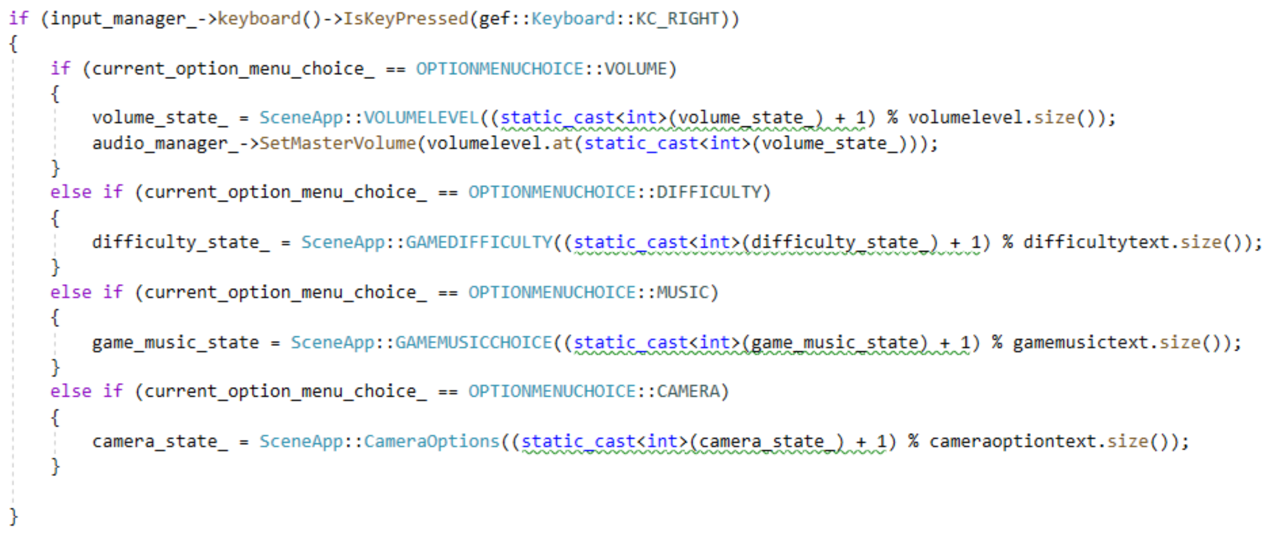
Enumerators and vectors were used to create a versatile menu system. Enums were created for a number of different states “Startmenuchoice”, “GameMusicChoice”, “OptionMenuChoice”, “GameState”, “GameDifficulty”, “VolumeLevel” and “CameraOptions”

Depending on the users input choice, for example, if the user pressed the down arrow key on the keyboard, the code would retrieve the current option or menu choice(depending on what game state we were currently in) and then increment that specific state which was then linked to the font->RenderText() function which handles the colours of the fonts by the variable:

```
font_->RenderText(
    sprite_renderer_,
    gef::Vector4(platform_.width() * 0.5f, platform_.height() * 0.5f - 56.0f, -0.99f),
    1.0f,
    textcolors[current_start_menu_choice_ == STARTMENUCHOICE::START],
    gef::TJ_CENTRE,
    "Start");

font_->RenderText(
    sprite_renderer_,
    gef::Vector4(platform_.width() * 0.5f, platform_.height() * 0.55f - 56.0f, -0.99f),
    1.0f,
    textcolors[current_start_menu_choice_ == STARTMENUCHOICE::OPTION],
    gef::TJ_CENTRE,
    "Options");
```

If the user then presses the Left or right arrow inside the options window, the text is decided using the code:



```
if (input_manager_->keyboard()->IsKeyPressed(gef::Keyboard::KC_RIGHT))
{
    if (current_option_menu_choice_ == OPTIONMENUCHOICE::VOLUME)
    {
        volume_state_ = SceneApp::VOLUMELEVEL((static_cast<int>(volume_state_) + 1) % volumelevel.size());
        audio_manager_->SetMasterVolume(volumelevel.at(static_cast<int>(volume_state_)));
    }
    else if (current_option_menu_choice_ == OPTIONMENUCHOICE::DIFFICULTY)
    {
        difficulty_state_ = SceneApp::GAMEDIFFICULTY((static_cast<int>(difficulty_state_) + 1) % difficultytext.size());
    }
    else if (current_option_menu_choice_ == OPTIONMENUCHOICE::MUSIC)
    {
        game_music_state = SceneApp::GAMEMUSICCHOICE((static_cast<int>(game_music_state) + 1) % gamemusictext.size());
    }
    else if (current_option_menu_choice_ == OPTIONMENUCHOICE::CAMERA)
    {
        camera_state_ = SceneApp::CameraOptions((static_cast<int>(camera_state_) + 1) % cameraoptiontext.size());
    }

}
```

By taking in the current enum state, and then using that to get the location of the vector element: std::vector<int> volumelevel = { 0, 25, 50, 75, 100 };

Strings were handled the same way, but using %s inside the RenderText() function

By using “enum class” instead of “enum”, the programmer was required to turn any calls from those enum classes into integers using a static cast.

For handling the players movement, Box2D ApplyForce() function was used. The player has no control over the “Ninjas” speed. An if check was required to be added and the applyForce function was only called if the Ninja's linear velocity fell below 7.0f.

For handling the players movement when the user presses W to jump, a burst of linear velocity is added to the y axis and depending if the y value is great or lower will then play the correct jumping or falling animation.

When the player is sliding, due to the animation and rotation, then an extra matrix was require to translate the body of the ninja slightly negative on the X axis to stop the appearance of the character teleporting.

For calculating the score, by using the GetplayerVelocity().x, the programmer was able to make the game logic of “if the player is moving faster than 4.0f, add to his score, if its below 4 then deduct from score”. When the ninja collides with one of the obstacles his velocity is reduced which then evokes the score being deducted. Originally, once the player's score dropped below a threshold, an audio cue would play. This was changed to work with the collision class. Now when a player interacts with an obstacle object, a random negative sound effect is played.

## 4.0 User Guide

The application is entirely run using the keyboard due to having no access to the hardware with the university campus shut down to the coronavirus lock down. The assessment was changed to allow keyboard input instead of PSVita input.

The return/enter key is used to select the main menu choices and the back option in the options screen.

Inside the options screen, the user will use the up and down arrows to navigate and then use the left and right arrow to select the multiple choice available. The user does not need to hit Return to “lock in” these choices.

During gameplay, the only two buttons available to the user is keyboard buttons W to jump and S to slide. Please hold this down until you have cleared the obstacle and then release.

For debugging purposes, press E during the main game loop to skip to the end screen

On the end screen your score will be displayed along with pressing Return/Enter to start a new game with the same choices, or you can press Escape to return back to the main menu to select

## 5.0 Data Oriented Design

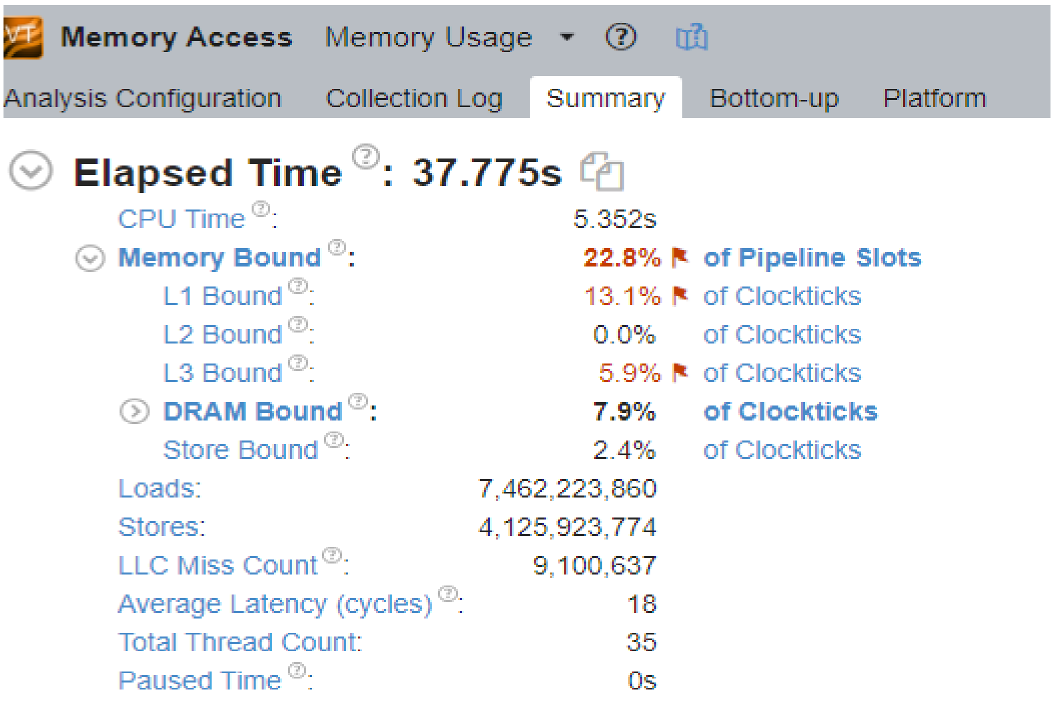
Data Oriented Design is a new way of thinking for current Abertay students. Since the first year, Object oriented programming is taught. Data oriented design is taking the focus away from mapping solutions to the code and more that solutions map to the needs of the hardware and what data “belong” together.

With this in mind, the current class structure of the code could be changed. With having a number of classes doing the same thing; meshinstances, initializing vectors and matrix and then transforming Roads,Buildings, background and obstacles, with the exception these also use B2B elements, by making 1 class which would use multiple creating std::vectors of data; sceneobjects{}, meshinstances{}, Vector3{}, Matrix44{}. Looping  through these vectors is a much more efficient use of memory as all the data is now contiguous.

The next thing that could be improved using DoD would be removing the use of hidden states in the scene_app if statements. Taken from Chris Trewartha's input on cache line misses, a bool used in an if statement could be an inefficient use of the cache by potentially taking up space and pushing variables that would have normally fit into the 64 cache line into a new separate cache line costing unnecessary cycle time.(1)

A big advantage to DoD especially with CPU's core count constantly increasing is the benefit of multiple cores and parallelization. The issue  around OOP is when trying to parallelize your code, you will often end up with multiple threads sitting idle waiting for others to finish. Using DoD we cut out a lot of the synchronization requirements  using large class structures(2)

With the project being changed from the Psvita to PC, the constraints on hardware changes considerably. If the current code was ran on the PSVita a more optimized DoD structure would be beneficial to performance. Here are the current cache misses for a short section of my gameplay:



```
Elapsed Time: 37.775s
    CPU Time:                              5.352s
    Memory Bound:                          22.8%   of Pipeline Slots
        L1 Bound:                          13.1%   of Clockticks
        L2 Bound:                           0.0%   of Clockticks
        L3 Bound:                           5.9%   of Clockticks
        DRAM Bound:                         7.9%   of Clockticks
        Store Bound:                        2.4%   of Clockticks
    Loads:                         7,462,223,860
    Stores:                        4,125,923,774
    LLC Miss Count:                    9,100,637
    Average Latency (cycles):                 18
    Total Thread Count:                       35
    Paused Time:                              0s
```

## 6.0 Conclusion

The module CMP208 Graphics programming focused on a number of areas that most 2nd year students were unfamiliar with. The GEF framework was particularly exciting and daunting to delve into, as it was the first fully formed framework the students have had access to outside of SFML and OpenGL.

From my own experience of the module, debugging the GEF framework was a challenge in itself due to the large class structure already in place. The provided GEF API documentation helped greatly when trying to explore how to do a specific game mechanic.

With the previous modules, although I had learned how to use matrices I had never actually used any in large numbers. From the start of the module, we were forced to move gameobjects/meshes using matrices. From my current knowledge, I feel this is one that has the most impact on general game mechanics.

The students had the chance to learn how the PSVita touch panels work during one of the labs. It was originally a project requirement, but with the coronavirus situation this was removed due to lack of available hardware.

Although 3d models were not a module requirement, with my large number of models and issues with the model loader provided, this lead to a number of issues that forced me to learn some knowledge with Maya which was explained above.

I had known about enumerators since first year, however, I have not used them to much effect until this module and in turn, creating a menu system using enumerators was an interesting learning experience.

Finally, this was the first time students in this course were using a physics system for their coursework through Box2D. With physics being such an integral part of game development, this experience i'm sure, will be invaluable.

## 7.0 References

1. Acton, M. (2014). Data Oriented Design and C++ [Online] Available at: https://www.youtube.com/watch?v=rX0ItVEVjHc [Accessed April 15th 2020]

2. Llopis, N (2009) Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)[online] Available at http://gamesfromwithin.com/data-oriented-design [Accessed April 14th 2020]

Art assets and models taken from Humble Bundle's " Best of POLYGON Game Dev Bundle "

Music taken from Youtube:
Easy Breezy - https://www.youtube.com/watch?v=76sNmqMzUuI
Run boy Run - https://www.youtube.com/watch?v=lmc21V-zBq0
Miami nights - https://www.youtube.com/watch?v=vC42MRmiAt0
Battlefield 2 - https://www.youtube.com/watch?v=zNd62SehRKY
End song – unable to locate online, video in media folder

Menu sounds and score announcer taken from DDR sound files from http://stepmaniathings.com/