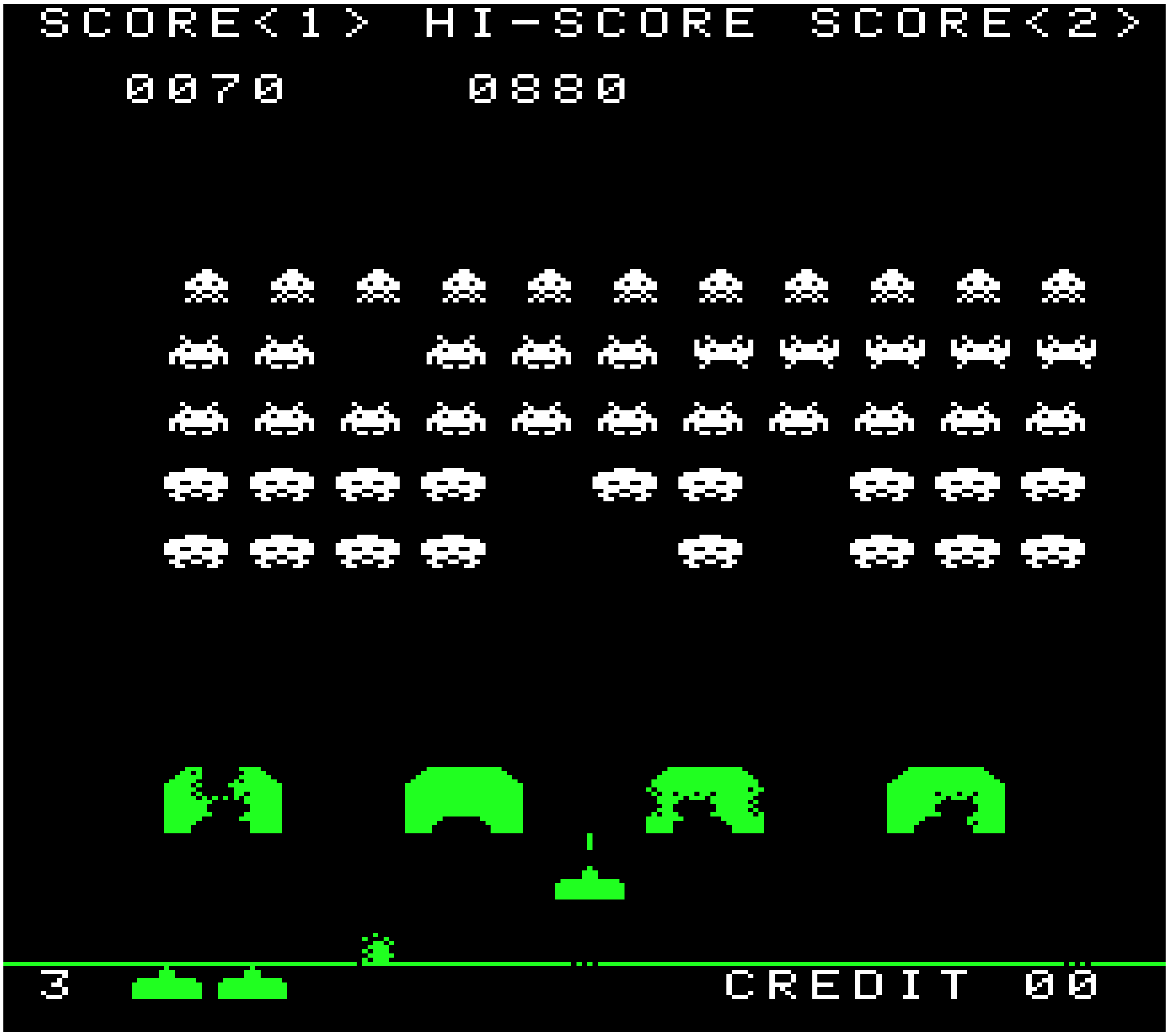
Kevin Keith
COSC 3P98 Final Project
Space Invaders
Brian Ross



How to Play/Run

1. Open latest version of XCode on Mac
2. On open, set signing to personal account if signing issue occurs
3. Hit build
4. Press left button to make tank go left
5. Press Right Button to make tank go right
6. Press Space to fire button
7. Press Esc to Escape the application
8. When enemies reach the end or you defeat all enemies it activates a game over function is called and you get your score

Project Background and Overview

NOTE - talk about initial purpose of the project, and then the “shift”

The purpose of the project was to take the classic video game from the 1990’s that ran on MS DOS - *“Scorched Earth”* and adapt the video game into a modern version that could be built and rendered using modern software engineering tools.

According to Wikipedia:

> *Space Invaders[b] is a 1978 shoot 'em up arcade game developed by Tomohiro Nishikado. It was manufactured and sold by Taito in Japan, and licensed by the Midway division of Bally for overseas distribution. Space Invaderswas the first fixed shooter and set the template for the shoot 'em up genre. The goal is to defeat wave after wave of descending aliens with a horizontally moving laser to earn as many points as possible.*

For this project, the Mac OS was chosen as the development and target platform using the xCode IDE (Integrated Development Environment), and the Swift programming language along with Metal to render the graphics.

The game would remain a 2-dimensional game given time constraints for the project.

Gameplay Overview

In the game, a wave of aliens come down from their ships and attack you. If they reach a certain point or you die 3 times, then it is game over. The goal is to get as high of a score as possible while avoiding the rain of bullets coming down from the enemy.

The game needs to track the hitboxes and movements for the missiles fired from both the enemy and the player to determine who gets hit.

Due to constraints both with the macOS and metalKit, and while there was an attempt to implement a main menu, however, it was decided that it would be better to just get the game working.

Desired Outcome

The main objective from this project is to deliver a basic working game where a player could play Space Invaders while learning Swift/MetalKit.

Metal vs. OpenGL

Metal kit is a shader library that uses .Metal Shader libraries to build a scene. Metalkit is a lot more complicated and difficult to get into due to having to learn how to use the metalRenderPipeline functions, command line functions, and setting them up so that you would be able to build any type of game that you are hoping for. What makes it good about openGL is unlike openGL, you are naming all of the meshes, variables and vertices that you end up using to create each of the builds, this puts you in a lot more control of your environment and how you use everything.

Challenges

Why the Original Approach was Altered

Originally the goal of this project was to build Scorched Earth using MetalKit. This was a game that appeared to be a relatively straightforward game to create. While Scorched Earth seems like a simple game, there are many complexities involved in creating it such as the types of weapons, randomizing the build of the terrain, factoring in wind as well as gravity.

With MetalKit and Swift being relatively new, and the another game appeared as an alternate option which could achieve the same objectives in:

- Building a simple 2-dimensional game with MetalKit
- Keeping the simple "retro" arcade aspect of the game in tact
- Demonstrating the capabilities of MetalKit and Swift
- Using the game engine built for this project and intended to be used with Scorched Earth
- Building a project of ample size that would allow a compare and contrast with OpenGL that was leveraged and used in class
- Remaining within the timeline provided to complete this project

With these objectives in mind, a switch of the game type for Space Invaders allowed the project to be completed and achieve the desired objectives.

Creation of Engine

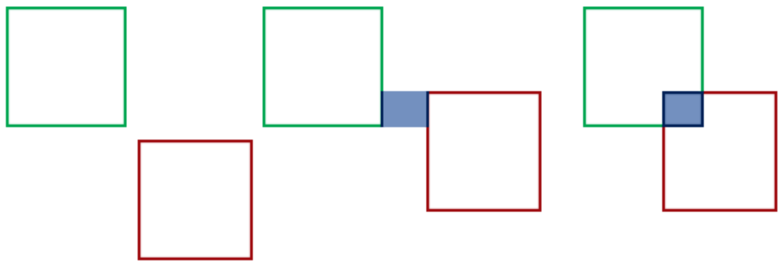
Due to the nature of information on MetalKit, the entire idea of this actually originally started off as following a tutorial for the creation/set up for a game engine created by 2eTime. Using ths, I was able to create the basic fragment and vertex shader, that I eventually edited into to create and add in textures, and read textures in as their own shape. This all culminated in myself eventually meeting up with 2eTime himself, who helped explain quite a few concepts, and organize my thoughts more into what I can do to build off of this. Specifically using multiple render passes to create an alternate texture that we can cull from the main one. Specifically, these meetings led to, "Mask-Fragment-Shader" and the final fragment and mask shader.

What changes needed to be made

The actual logic of the game itself, along with the 2-dimensional graphics to represent the "Space Invaders" and the "Player" were the only real change required in order to switch from one game to another. Space Invaders does not need to account for gravity or wind nor does it require the use of a variety of different weapons. The game elements which include:

- Objects that can shoot projectiles at each other
- Targets that could be hit leveraging a "hit box" and "pixel collision"
- Management of several different objects

In other words, the basic logic created was the same, it was how that "game engine" was used that was changed.



How Challenges were Resolved

- First challenge was learning and setting up the project
    - This was overcome by finding a community, and tutorial, of which I had used to set up my scene, and render pipeline/descriptor to build the bare minimum that I would need to build the game that I had wanted to build.
- The Second challenge was learning Swift:
    - This is not a complicated language to learn, as it’s a mixture between Python, Haskell, Java, and C++, taking all of the advantages from all of these. The changes are simple, but really fun to learn
    - The hard part of this was learning and using the new libraries to create the applications and using them in the applications.

The interesting Classes/Functions:

AABB Collision:

AABB collision is a fairly simple but still extremely useful technique in Graphics. Here we take the x, the y, and the width and the height to create a bounding box. These bounding boxes can be used for a variety of things, like creating platforms for an object. How AABB collision was used was as an event trigger that goes off if it detects either the player's bullet on an enemy or an enemies’ bullet on the player. If the box connects with the player, it sets off an explosion and sets the player at the middle of the screen if they have an extra life, if it hits the enemy, it removes the enemy from the list.

RenderPipeline and classes associated

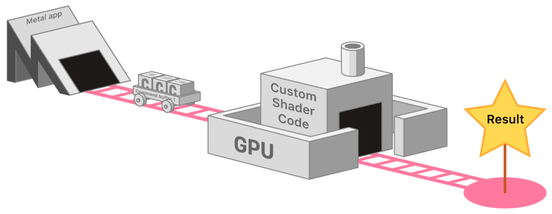
Command Queue:

This works as your typical queue, whatever command or object you tell it to will go off in the order that you told it to render in. This works as a First in First out, and tells the render what it will show on the screen first on the clear screen. Essentially we use this to create our objects and place them on the screen in Metal.

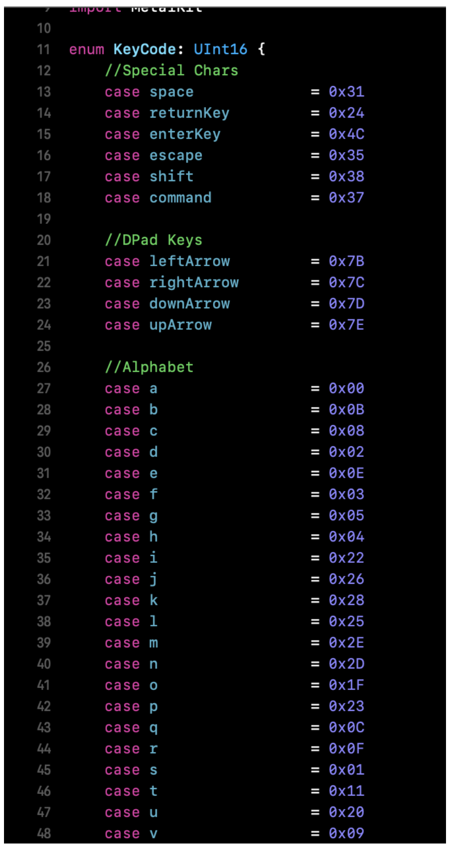
Here we get a much clearer view of doing exactly this, when we take a look in our Renderer class with all of the calls to the Command Queue and the Command Buffer class. You can find this in multiple of my classes for the render, as this is what allows the objects in my node/scene to render into the screen.

Multiple Render Passes:

This is extremely useful. We do this by creating a culling mask in our .Metal shader, for me it's the mask_fragment_shader function. Using this, we can create multiple renders, and create a mask for each of the items that are drawn onto the scene, like so. We can then go and create a texture of each of these items, place them on a quad, and render them on said quad during each frame.



Keyboard Press

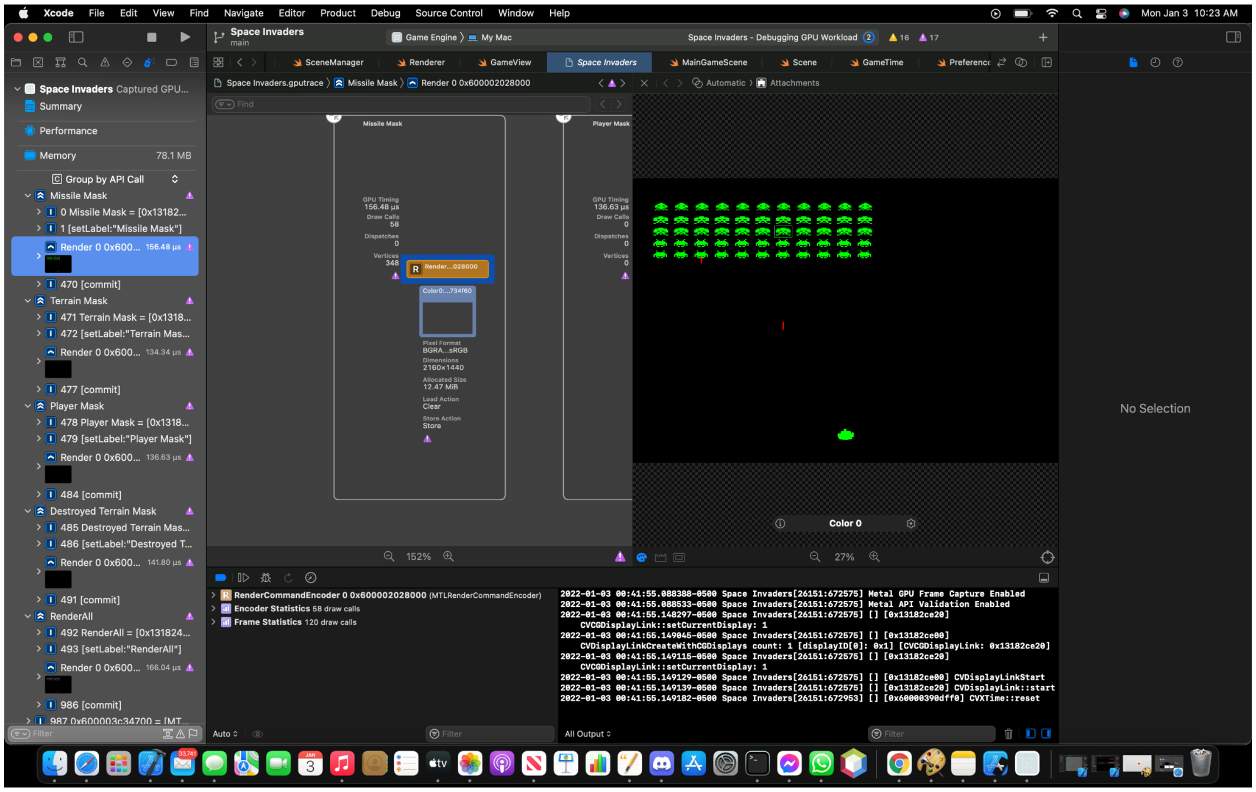


```
import MetalKit

enum KeyCode: UInt16 {
    //Special Chars
    case space              = 0x31
    case returnKey          = 0x24
    case enterKey           = 0x4C
    case escape             = 0x35
    case shift              = 0x38
    case command            = 0x37

    //DPad Keys
    case leftArrow          = 0x7B
    case rightArrow         = 0x7C
    case downArrow          = 0x7D
    case upArrow            = 0x7E

    //Alphabet
    case a                  = 0x00
    case b                  = 0x0B
    case c                  = 0x08
    case d                  = 0x02
    case e                  = 0x0E
    case f                  = 0x03
    case g                  = 0x05
    case h                  = 0x04
    case i                  = 0x22
    case j                  = 0x26
    case k                  = 0x28
    case l                  = 0x25
    case m                  = 0x2E
    case n                  = 0x2D
    case o                  = 0x1F
    case p                  = 0x23
    case q                  = 0x0C
    case r                  = 0x0F
    case s                  = 0x01
    case t                  = 0x11
    case u                  = 0x20
    case v                  = 0x09
```

Unlike OpenGL, we have to go and call out each of the Keyboard buttons and assign it's assigned Hexadecimal value onto it. This means that we absolutely have to create an enumerator with each of the values onto it.

Metal Debugger

This is by far the most helpful tool that Metalkit allows. When you open up the Metal Debugger it places everything that you are using into the command render, and renders everything on top of one another. This is extremely helpful when you are doing something like say, creating a culling mask. This allows us to see all of the bound resources, and can even help us figure out an issue with the code. For example, early in development, when I was working with texturing, I made a mistake of calling the texture after I sent it through the command buffer. This caused a big black screen to show up where I needed the quad. I managed to debug the issue using the metal debugger as the api commands show me that while I am loading the texture, the texture was not getting bound to any object.



```
Space Invaders Captured GPU...
  Summary
  Performance
  Memory                           78.1 MB
  Group by API Call
  Missile Mask
    0 Missile Mask = [0x13182...
    1 [setLabel:"Missile Mask"]
    Render 0 0x600...   156.48 µs
    470 [commit]
  Terrain Mask
    471 Terrain Mask = [0x1318...
    472 [setLabel:"Terrain Mas...
    Render 0 0x600...   134.34 µs
    477 [commit]
  Player Mask
    478 Player Mask = [0x1318...
    479 [setLabel:"Player Mask"]
    Render 0 0x600...   136.63 µs
    484 [commit]
  Destroyed Terrain Mask
    485 Destroyed Terrain Mas...
    486 [setLabel:"Destroyed T...
    Render 0 0x600...   141.80 µs
    491 [commit]
  RenderAll
    492 RenderAll = [0x131824...
    493 [setLabel:"RenderAll"]
    Render 0 0x600...   166.04 µs
    986 [commit]
  987 0x600003c34700 = [MT...
```

```
GPU Timing 156.48 µs
Draw Calls 58
Dispatches 0
Vertices 348

GPU Timing 136.63 µs
Draw Calls 0
Dispatches 0
Vertices 0

Pixel Format BGRA...sRGB
Dimensions 2160x1440
Allocated Size 12.47 MiB
Load Action Clear
Store Action Store
```

```
RenderCommandEncoder 0 0x600002028000 (MTLRenderCommandEncoder)
Encoder Statistics 58 draw calls
Frame Statistics 120 draw calls
```

```
2022-01-03 00:41:55.088388-0500 Space Invaders[26151:672575] Metal GPU Frame Capture Enabled
2022-01-03 00:41:55.088533-0500 Space Invaders[26151:672575] Metal API Validation Enabled
2022-01-03 00:41:55.148297-0500 Space Invaders[26151:672575] [] [0x13182ce20]
    CVCGDisplayLink::setCurrentDisplay: 1
2022-01-03 00:41:55.149045-0500 Space Invaders[26151:672575] [] [0x13182ce00]
    CVDisplayLinkCreateWithCGDisplays count: 1 [displayID[0]: 0x1] [CVCGDisplayLink: 0x13182ce20]
2022-01-03 00:41:55.149115-0500 Space Invaders[26151:672575] [] [0x13182ce20]
    CVCGDisplayLink::setCurrentDisplay: 1
2022-01-03 00:41:55.149129-0500 Space Invaders[26151:672575] [] [0x13182ce00] CVDisplayLinkStart
2022-01-03 00:41:55.149139-0500 Space Invaders[26151:672575] [] [0x13182ce20] CVDisplayLink::start
2022-01-03 00:41:55.149182-0500 Space Invaders[26151:672953] [] [0x60000390dff0] CVXTime::reset
```

Moving forward:

If given the opportunity to work on a project like this, either for a COSC 3P99 or COSC 4F90, I would do something more akin to the ray tracers, with development of the application being more in Objective C in Metal rather than Swift, to be more relevant to today's standards in regards to the application. Using opportunities like this to further my knowledge on the Metal Library and Apple App Development in general.

References:

Learn OpenGL - Collision Detection
(https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection)

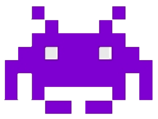
Apple Developers - Command Buffer

https://developer.apple.com/documentation/metal/setting_up_a_command_structure?language=objc

Wikipedia - Space Invaders image

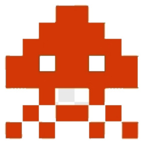
https://en.wikipedia.org/wiki/Space_Invaders

Textures:



https://www.clipartmax.com/middle/m2i8A0b1K9K9H7K9_space-invaders-3d-pixel-art-space-invaders-pixel-art/

https://www.clipartmax.com/middle/m2i8A0b1K9K9H7K9_space-invaders-3d-pixel-art-space-invaders-pixel-art/



https://www.clipartmax.com/middle/m2i8A0b1K9K9H7K9_space-invaders-3d-pixel-art-space-invaders-pixel-art/

https://www.kindpng.com/imgv/bhbmh_transparent-background-cartoon-explosion-hd-png-download/

https://www.deviantart.com/revraptor898/art/Space-Invader-Tank-187733221

2eTime’s YouTube:

https://www.youtube.com/c/2etime/videos