Connor Chow

Trevor Tomesh

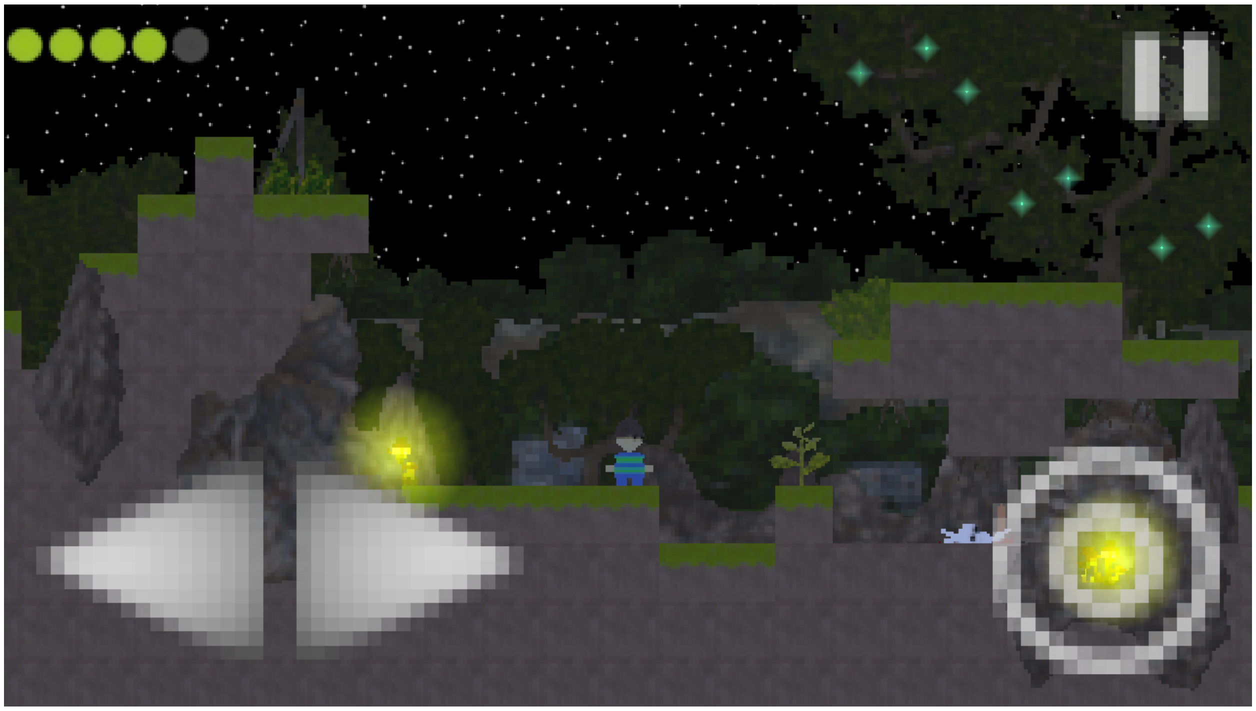
COMP-2430-WDE

22 April, 2021

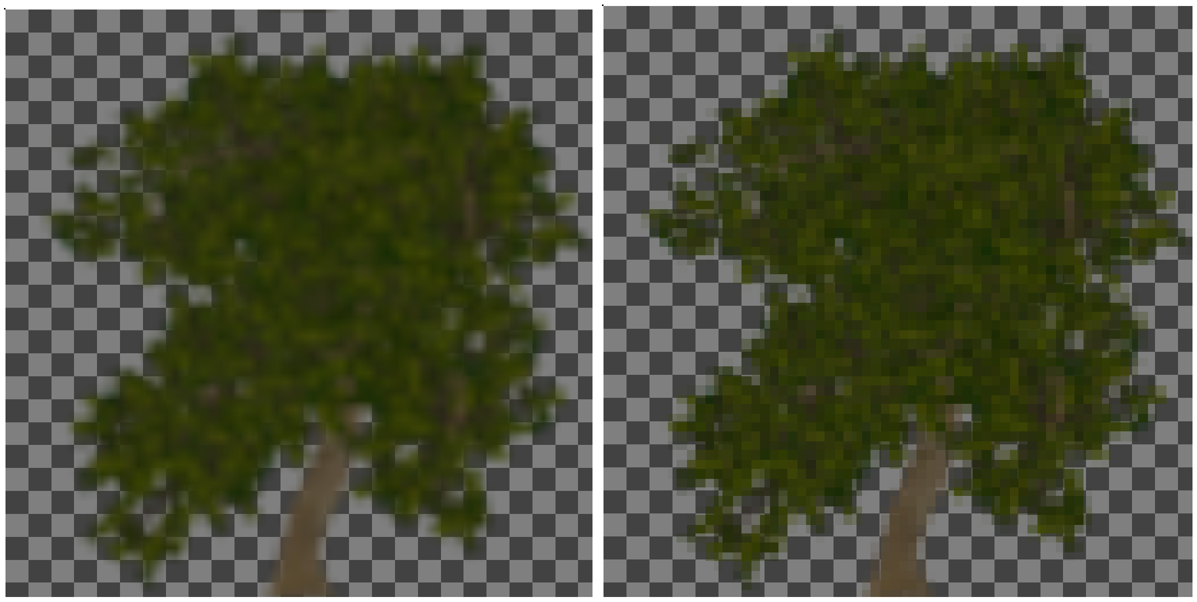
Working with Unreal Engine 4 to Produce a Mobile Phone Game

After much experience with 3D development in Unreal Engine 4 (UE4), I was considerably excited to spend over a month dipping my toes into UE4's 2D features for something school-related. I thought that with how intricate and functional many of UE4's 3D features were that 2D would be a breeze. Well after the past few weeks developing my "Into the Mystic" video game in 2D for Android platforms, I've decided that UE4 is probably one of the last engines I should make a 2D game in. Many of the features that were useful in the 3D realm tended to be borderline broken when flattening things out, and I don't think this engine was ever designed to support an infinite side-scrolling open world. Allow me to delve into the many triumphs and struggles I experienced on a weekly basis.

Week 1 (March 18 - March 25)



This week was primarily spent making 2D sprites in blender, exporting them to unreal, and creating the basis for randomly generated terrain. One nitpick is that with my pixel art the results look amateur at best. One problem I attribute to this is the ground, which is a bunch of square blocks that frequently has line gaps on it, and the second biggest problem I saw with the visual details was the pixel density. It was never consistent. My workflow involved making 16x16, 32x32, 64x64, and 128x128 textures for each model and importing them into unreal with minimalized texture filtering to give off a pixelated look. I did not have any proficiency in hand-painted pixel art so this was my best bet at pulling off a world that vaguely resembles something. The below images show a frame with compression on the left, and the same frame without compression on the right:



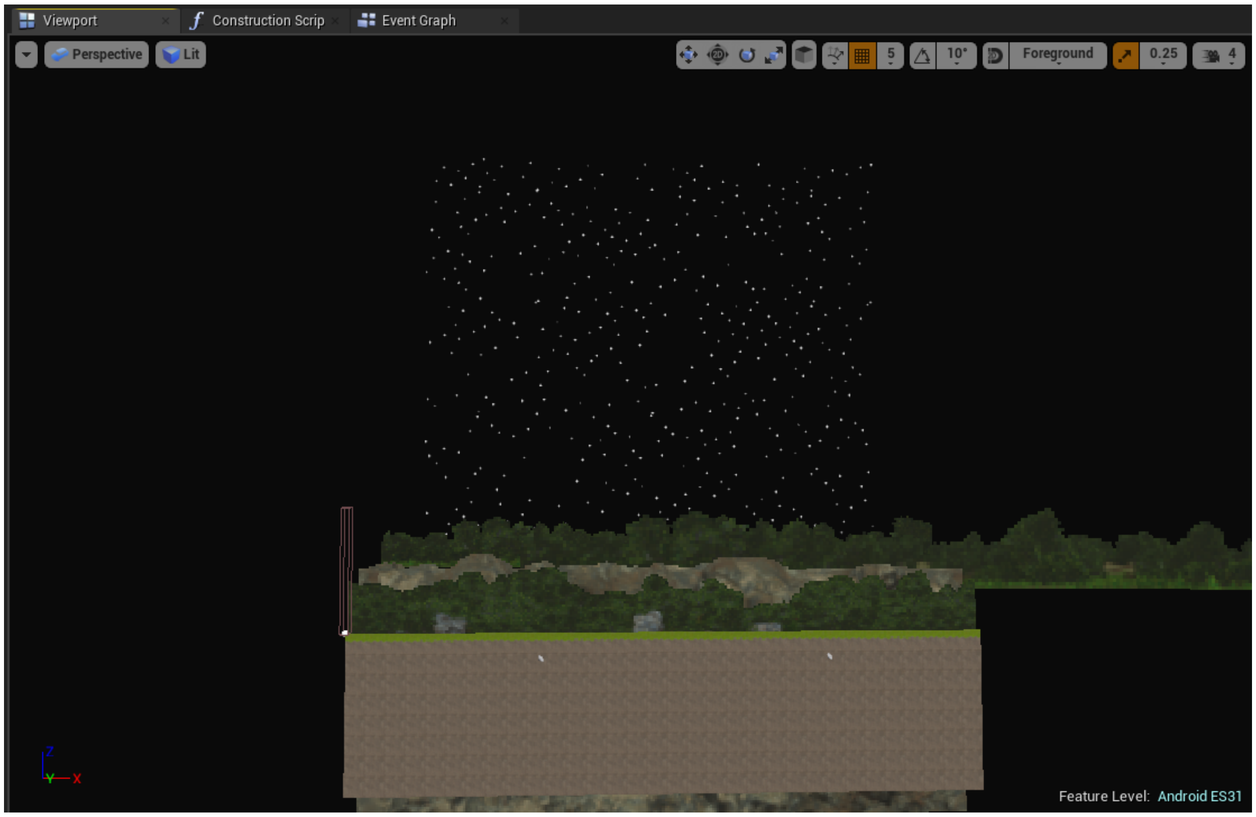
As is clearly visible, the texture with UE4’s default filtering looks blurry while the texture that uses minimalized filtering options has the desirable pixelated effect I was going for. While this looked great on my desktop, it had some unsightly compression errors when I ran a test apk of the project. After a few hours worth of troubleshooting I found out that I had to change the compression settings to “user interface” in order to make sure part of a window wasn’t missing and there weren't artifacts all over the player’s body. This was somewhat irritating however, since there’s no straightforward way to modify the default import settings so I had to change these after importing each texture.

Everything animated is a paper flipbook, which consists of a couple textures put together into a cohesive animation. For the most part these frames were done in blender except for the death animations on characters which were hand-drawn using GIMP photoshop, to turn each creature, including the player, into a bursting blood sack upon death. For characters and such I used an integer variable called animState which serves as an index to show which flipbook is being used at one time.

I think most of these problems could have been avoided if I took all of my static sprites and stuffed them into a bigger texture (say a 1k texture rather than a couple dozen 32-bit textures). This probably would have been friendlier on performance and harmonized the pixel density on each asset. I had no reference of other assets while making the individual rocks and such and so I did the worst thing possible: I just guessed for the most part what would look good. While it’s too late to change that now, It is a good thing to keep in mind if I ever try my hand at pixel art again.

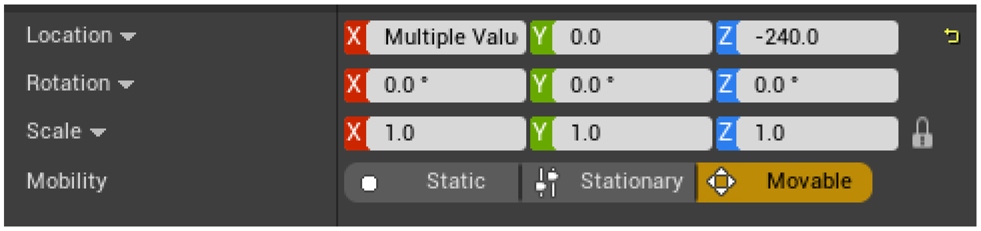
As for the ground, the tiles were very functional for movement, but I noticed that the slightest displacement, which was stupidly easy to do in the actor editor, caused unsightly looking lines that required me to manually go to each individual tile and fix it’s location. This was heavily frustrating due to the workflow required to generate my desired kind of open world.

In order to discuss the next aspect to the best degree I should cover some of my preferred tools in the ue4 editor that makes world generation easy and fast. When you have an asset selected in the editor you can hold the ALT key and move the asset using the editor’s directional arrows, rotation orbits, or scale; and you will duplicate the object and have the instant ability to move it wherever you want. However, I can’t just assemble an open world in the editor since I want terrain to be randomly generated. So I had to assemble my terrains in the separate actors to break them up into sections, since it allows me to use different behaviours that give each section the ability to spawn new terrain right next to it when the player steps into it, allowing infinite world generation. Allow me to illustrate:



This is the default terrain section for outdoor areas. It is its own actor entity in the world that can be instantiated at any point thanks to a bit of spawn logic. The pink box on the right is an overlap collider that fires whenever the player touches the object. This collider is an instance of another actor which holds all the logic. It has a few customizable settings that I programmed into them using blueprints. The first option is the default array, which is an array of other terrain pieces that can be created next to this given terrain. It will pick at random which terrain it will spawn next. The next option is generating secrets. This is a boolean that decides whether this terrain piece has a chance of generating a piece of terrain with story pieces or not. If true there's a 20% chance that a secret will be generated, and when it generates a secret there is a 20% chance that the player will get a story piece with the Dishonoured Elf character (which had not yet been created) or encounter a small or big village of Redcaps that could have runes detailing their history (still looking deserted). At this point there is not much variety yet, but more gets added on as I continue developing features, and with this relatively freeform structure I was able to generate a lot on-the-fly, despite its many drawbacks. This is also how I generated caves, where a cave exit would be spawned with an overlap component that only spawns cave corridors, troll caves, and cave exits. Cave exits only can go back to spawning forest terrains or potentially more caves. So all I needed was a starting area in the default game level which also served as a transition into the first procedural actor. So from there I could spawn worlds infinitely.

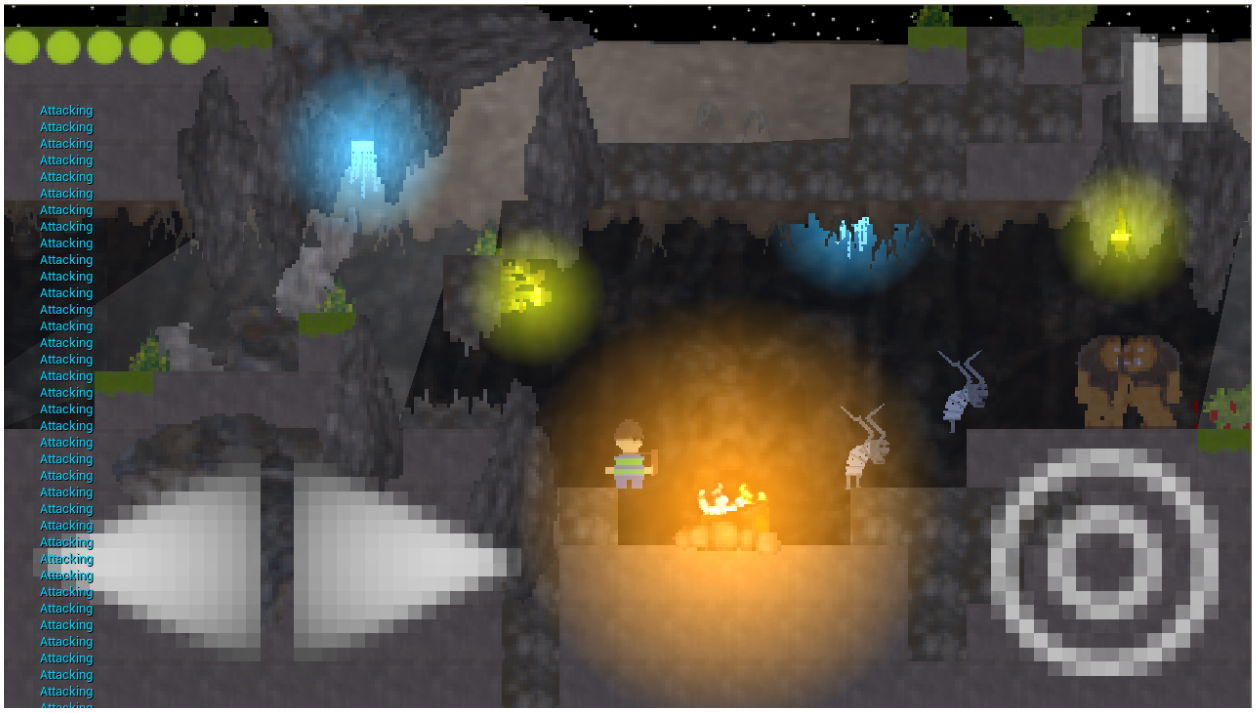
Unfortunately using actors to assemble the bulk of my world began to rear its ugly head almost instantly. Say you have a row of 52 grass tiles, like in my instance. If this was in the world editor I could border select them and hold the ALT key and use the location arrows to pull it down 160 units to flesh out the ground underneath. Then with everything still selected I would disable colliders and transform the grass block into dirt. After that I would just keep duplicating and moving it down with the ALT key. Quick and simple. Unfortunately the actor editor in UE4 does not accept this workflow. So I have to manually select everything with the left-mouse button, and if I hold it down for a millisecond too long the actor editor tries moving what I have already selected, which means I have to undo, which deselects everything and forces me to start over. I should also mention that this entire time the actor editor is freezing up on me since it runs slower than the regular world editor. Next I press CTRL+W to duplicate the terrain which deselects everything except the first or last tile I selected. So I go back to selecting the entire row I just duplicated while hoping to god the actor editor doesn't randomly freeze up on me, losing all my christian faith in the process. After everything is selected I can't really use the direction arrows on the assets because the actor editor freezes up to the point where it won't let me see what I am doing until I release the left-mouse button. That meant that I had to manually transform things using the details panel. You also can't view collision boxes in the editor so the only way to actually make sure the player doesn't fall through the world or get blocked by a background prop is trial by fire.


Location X Multiple Valu Y 0.0 Z -240.0
Rotation X 0.0 ° Y 0.0 ° Z 0.0 °
Scale X 1.0 Y 1.0 Z 1.0
Mobility Static Stationary [✓ Movable]

Notice how on the x-axis it says "Multiple Values." This is for a row of tiles and means that if I apply any location changes on the x-axis, all the tiles selected go to the same x-location, destroying the row I just made. Fortunately I usually only needed to move rows along the z-axis and columns along the x-axis, but this limitation still proved to be a massive problem when selecting and moving large blocks that were more than one block wide and one block tall. In those instances I was only able to run off of pure guesswork which was not fun.

If I had allowed more time with world generation I probably would have gone with something more procedural, using perlin noise and other random generation methods to randomly create mountains, forests and caves similar to games such as Minecraft or more related 2D games like Terraria. However I also wanted to implement a stealth system and three factions of NPC's who fought the player and each other, which took up a large bulk of the next few weeks.

Week 2 (March 25 - April 1)



```
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
```

Before delving into week 2 I should talk about the commenting system in UE4's blueprints. It's messy. Blueprint nodes, which dictate logic and behaviour, can be enveloped with a big white box and labelled in wide text. It's okay for separating larger functions and behaviours, however after commenting out the important information my blueprint got really bloated really fast with text and boxes inside boxes and I had to spread things out considerably to make sense of it all. I would have preferred if these comment boxes didn't show text until you hovered over them, and if nested boxes would override the text of parent boxes. Also comment boxes might look like they've fully enveloped your function, but the scale of the boxes and the nodes changes when zooming in and out and if for some reason the editor thinks a node is even a smidge out of the comment box that node will be left behind when you try to move your function or whatever.

Week 2 was when I began the daunting task of creating AI that properly attacked the player and each other as well as cheap lights that calculated nice effects in large densities upon spawning that didn’t melt peoples’ phones. I was hoping that UE4 had 2D features which would allow me to implement AI to a similar degree of effectiveness as in the 3D world. However, Behaviour Trees require a navmesh in order to locomote effectively and utilize many of its systems, which did not translate even remotely to the 2D world space. Navmeshes require a terrain that a pawn can fit itself onto on the X, Y, and Z axis. This would have worked on the X and Z axis, however ground tiles were paper thin, making the Y-axis invalid. Not to mention I could not find an efficient way to generate a navmesh on an infinitely generated open world. So what I did instead was heavily modify the basis of the player character in order to create an NPC that could autonomously execute its own decisions and actions. Albeit these behaviours and actions are pretty janky. One notable problem was the stealth system. Trolls and ghouls could see enemies and the player through walls because of the paper-thin colliders, which made the experience often feel unfair. The player could hide in bushes though, which made them invisible to the NPCs surrounding them. I arranged the Y location of the player, the bush, and the NPCs so that the player would be behind bushes and the enemy NPCs would be in front of bushes. When the player is hiding behind bushes they auto-crouch and are immune to damage and detection.

For detection I use three key variables: Detection Cache (integer), Detection State (integer), and infighting (bool). In concept the AI has three states which establishes the foundation of their rules: idle (DetectionCache < 1, DetectionState = 0, infighting = false), alert (0 < DetectionCache < 4, DetectionState = 1, infighting = false), Attacking Player (DetectionCache >= 4, DetectionState = 2, infighting = false), and infighting (infighting = true). As you can see from the logic above, enemies will prioritize other enemy factions over the player. Redcaps will fight Ghouls and Trolls, Trolls will fight Redcaps and Ghouls, and Ghouls will fight Trolls and Redcaps. During week 2 I only finished programming the Redcap, but other enemies were very similar in functionality and design, and were quickly churned out after I finished my base design. The only thing I really changed up was health, flipbooks representing the characters, and attack types. The way these guys detect enemies is using the pawn sensing component in ue4, which gives the enemy a detection cone and distance at which they can see pawns. If the player is in sight then the detection cache will increment as long as enemies are not infighting and as long as the player isn't hiding behind a bush. If the player gets out of sight the enemy's detection cache will start decrementing down to 1 but it will remain alert and go to investigate your last known location. They will come visit and if they do not notice the player again they will return to their post and begin idling again. If the player is right close to the enemy and they get spotted they are instantly spotted. The enemy now enters the attack stage and alternates between charging, retreating, and running its attacks on the player.

On death the enemy explodes and I have a timer set up to terminate the actor once the death animation finishes running. I do not like how you cannot insert functions into flipbook animations that get fired once reaching a given frame, similar to how 3D models work. It forces me to rely on delays and counters to execute behaviours appropriately in accordance with the animations, and sometimes the results are not great.

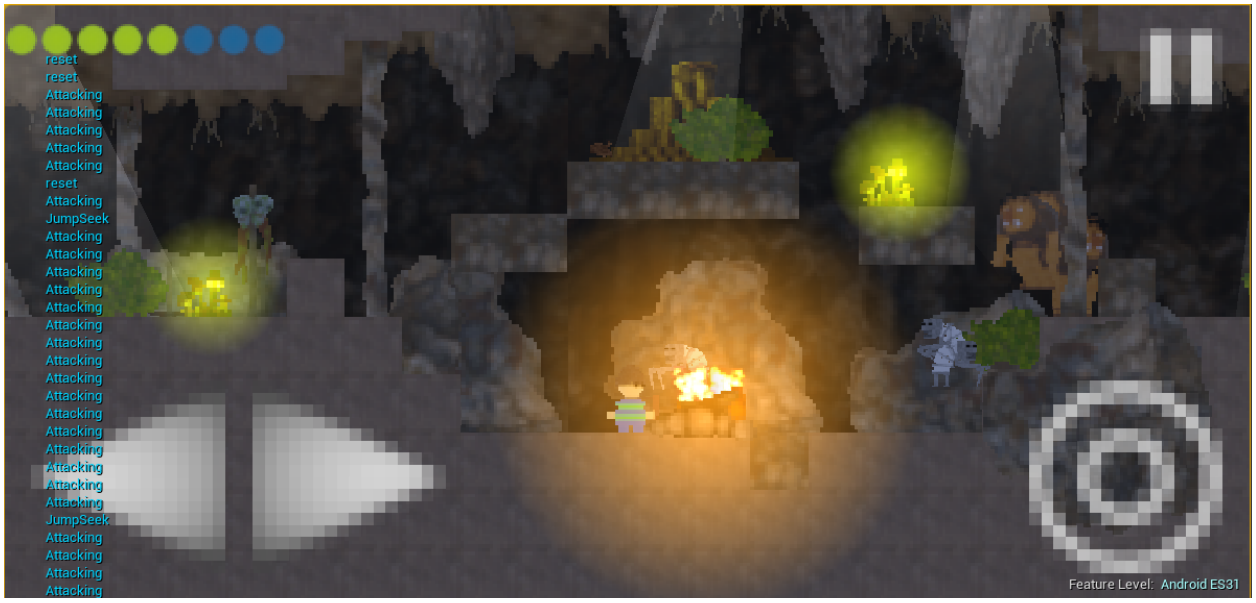
To account for other detection-related instances I placed a collider behind the AI's back that triggers whenever a pawn overlaps it. When triggered the AI will spin around and if they see nothing they will return to idling. The AI will also be able to recognize when the player hits them and when something that isn't the player hits them.

For infighting the enemies have two separate enemies they can attack, so depending on which enemy-type they spot they will have an index which is set accordingly to the reference they capture. After a few seconds of not being able to see the enemy the AI will lose interest. I then use a switch statement to dictate which attack behaviour is exhibited, so that the enemy isn't trying to attack nothing and just standing there. So say a troll spots a ghoul while fighting the player. The infighting for the troll becomes true, so he loses interest in the player's existence. The troll can now either set this ghoul they spotted to closest enemy ghoul with an enemy index of 1, or closest enemy redcap with an enemy index of 0.

So with this complex implementation I have a mostly working stealth system that allows players to utilize a bit of sneak-around and take advantage of systemic events.

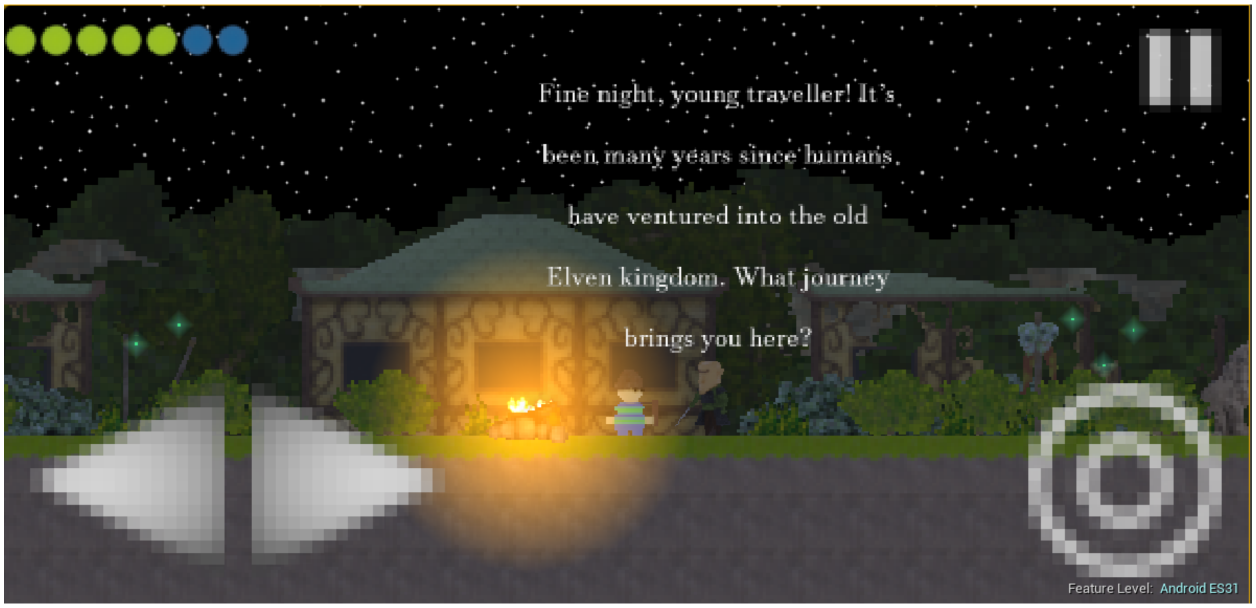
One of the most vital pieces to creating these systemic scenarios as well as making sure the player was not getting brutalized was spawning. Despite the world being random, I can still take advantage of the customizable tools to make enemies run right into each other. For example ghoul caves do not usually spawn by default. Instead they only spawn after troll caves and always send a party of ghouls who by default idle by walking to the left. Trolls can also randomly leave to forage by walking towards the right, meaning they are destined to collide with enemy factions. Troll caves are designed to either spawn a regular exit, an exit with a small ghoul lair, or a ghoul lair (which can only spawn an exit). That way the player doesn't run into a whole sequence of troll caves or ghoul caves and get obliterated. Redcap huts are also not able to generate further secrets, meaning the dishonoured elf storyline won't randomly appear after fighting redcaps, and the player hopefully shouldn't experience a terrible bad luck with several redcap villages in a row.

Now for something that doesn't give me a headache to explain. The lights. Originally I was planning to use dynamic point lights for the engine but unfortunately UE4 mobile only supports up to 4 dynamic lights at a time, and as you can see I like to fill my scene with many lights. Instead I went for a cheaper, material-based lighting method with a procedural mesh. The mesh uses a line trace by channel and a for-loop to draw a sphere until it hits a collider and stops. I then use some falloff math on the UVs that extend from the center outwards so that when I create an additive material for it, use a gradient mask node on the texture coordinate and multiply its inverse by the emissive colour and opacity, the mesh creates this nice falloff effect where it's the brightest in the center and fades on distance. I also made a glowing animation using time and cosine with a bit of math to ensure it was in the desired range (between 0 and 1) and multiplied that into the opacity field, then used some location based randomization to generate some offset in the glowing animations. While it doesn't actually light anything, it fakes an illumination effect on everything since the game is already globally lit with a nighttime-cold hue. With this procedural mesh I was able to fit it into cave crevices as well to fake some really nice-looking light shafts.



```
reset
reset
Attacking
Attacking
Attacking
Attacking
Attacking
reset
Attacking
JumpSeek
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
Attacking
JumpSeek
Attacking
Attacking
Attacking
Attacking
```

Week 3-4 (April 1 - April 15)

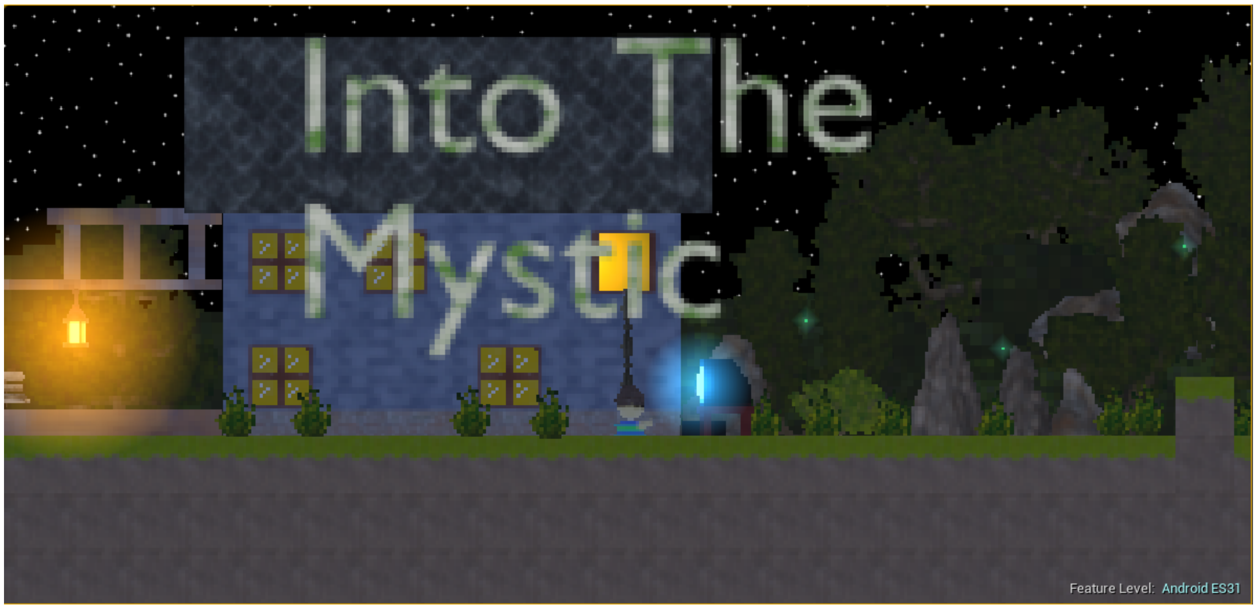

Fine night, young traveller! It's been many years since humans have ventured into the old Elven kingdom. What journey brings you here?
Feature Level: Android ES31

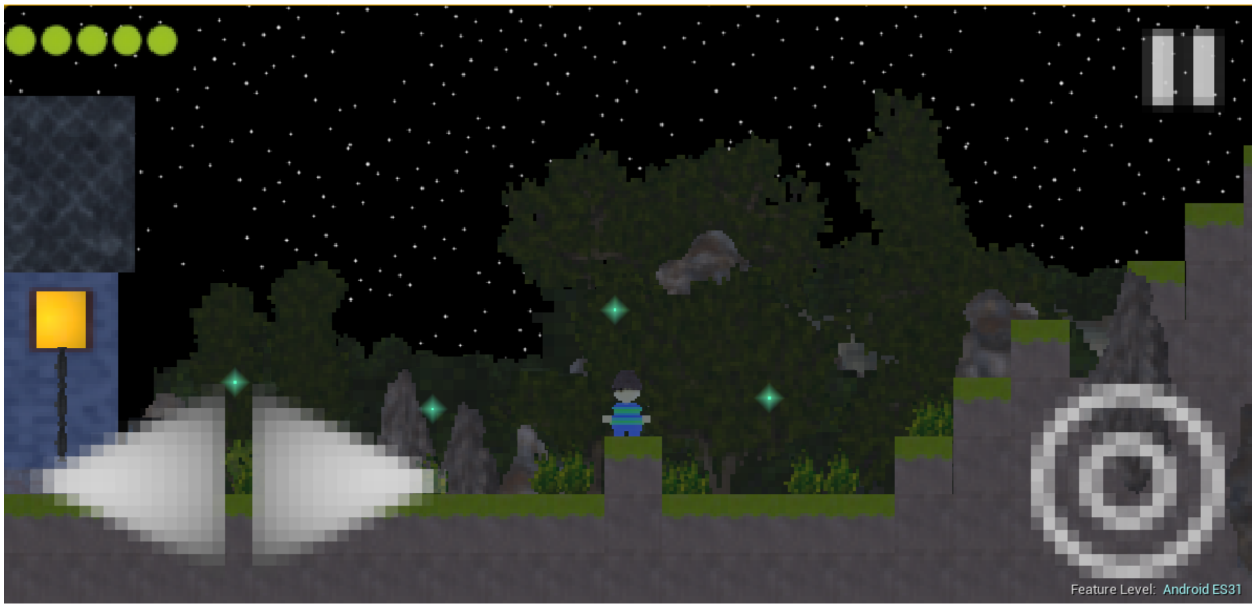
Before delving into pickups and interactables I took a considerable amount of time these two weeks to enrichen the world with an interesting character as well as a bit of history about the kingdom. The dishonoured elf is an old soldier that you meet twice in your journeys. The first time being in the ruins of an old kingdom, and the second time in a cave full of dead trolls. The redcap runes are the only historical texts that exist around the world. You can read about how they are rebuilding their civilization after the fall of the elven kingdom, the powers that they believe exist behind their traditions, and the occasional ghost story. However I didn't want the player to run into the same experience/conversation more than once every playthrough otherwise the player will get bored of the idea before they even run into the second encounter. So I have 2 save markers: a counter for the elf and one for the runes. Every time you finish reading a rune or finish a conversation the game will increment the save information. So if the elf is on 0 you will meet him in the ruins. If it is 1 you will meet him in the troll cave. If it is on 2 you will be discovering these interesting ruined cave halls. Making the game save was considerably more complicated than I had originally anticipated. It involves a lot of safeguarding and there is a hidden checkbox in the project settings you have to check if you want the save progress to work properly. However once I got that sorted out it was smooth sailing from there.

The next weekly item on my list was pickups and interactables. I had three actors I wanted to swap around interchangeably: armour pickups, health, and bushes. However, it wouldn't make sense for bushes to grow underground with no sun. So I created a spawner that traces a line by channel directly upwards. If the line hits something, it can only spawn armour and do it 25% of the time. If it doesn't hit something, it can either spawn a healing plant, a bush, or nothing. I liked this feature because it meant that caves served the purpose of providing the player with ample armour, while on the surface above they could search for healing plants and make better use of bushes.

I should also note that during week 4 I was finishing making the Troll, Redcap, and Ghoul. I implemented some optimization functions as well that disabled their actor tick (runs calculations every frame) at a certain distance. Most AI behaviours run on tick, which was not optimal for performance in hindsight, but I found that even with the bigger fights between trolls and ghouls the game was still running well on my android device. Another feature I implemented was the ability for AI to notify their nearby friends of the player or of another enemy. I did this by having them cast to all actors of the same class, which provides a reference to to instanced actors in the world, and update their friends' behaviour state with their own state if they are within a certain range. This was done on intervals so that the player still has a chance to lose them.

Week 5(April 5 - April 22)





This final week I spent finalizing the experience. I added a title page, a starting area for the player, and a TV which deletes it's light and switches to an off state when the player starts. I also programmed the player so that they are sitting down playing games on the TV before playing, and they start off unarmed before taking a slingshot from a bloody corpse they find. I drastically changed many of the in-game controls since I found the original controls to feel clunky in practice. I added a pause menu too, which I don't want to talk about. This is also where I implemented the health system and player death/restart which made me wish I got this over with sooner. I had to program health states for each level of health, as well as for the shields. Each digit of health had to be represented by a circular image on the screen. So say the player's health was 5 and then they took damage. The below table represents how each health field is affected:

Time					
Before Hit	Green	Green	Green	Green	Green
On Hit	Green	Green	Green	Green	Red
After a short delay	Green	Green	Green	Green	Grey

This transition was done using a timer delay node and swapping out the images that the health image is displaying. While relatively non-challenging, this was quite a time-consuming setup as I had to program the transition between each health change. This behaviour exists in update functions which only fire when the player takes damage or collects an item pickup

I made animations that made the player character itself flash red to show off the damage buffer, as well as for the enemy. I also added an animation that made the player character flash yellow when they picked up a healing plant.

My next focus was sound design. I created sound cues which hosted several different audio files and used some sound editing to create some unique sounds and also add variety. I then fired off my given sounds where needed in the blueprint editor. For example, say a troll dies. Once the animation starts I use a do-once node to prevent audio overlap, then use the simple node “play sound at location.” A key setting under this is the attenuation settings. It’s most basic purpose is to dictate how the sound behaves depending on the camera’s distance. I tweaked around with these settings on most props to achieve something that feels tactile and loud enough, but fades reasonably with distance. I also added music using the level blueprint, which would stop suddenly and play death music when the player died. I wish I left myself more time to polish the sound, but overall I think it is serviceable for what it needs to do.

Conclusion

There were a few things I didn’t quite pull off. I didn’t get the chance to implement crossbows or power-ups, but I don’t think the lack in these things waters down the overall experience. Most of my gameplay rides off of the idea that users are playing this to discover new scenarios and places and hopefully interact with the world and its systems in an interesting way. There are many great things about the unreal engine: its optimization features, which is largely culling objects out of view from drawing; its variety of tools, which in some areas enable me to make some truly interesting features; and its many visual graphical tools laid out for me. I hardly doubt I could’ve pulled off the same feat using android studio and I am pretty happy with the fact that I finally finished a game that isn’t completely broken. But I don’t think that Unreal engine is best suited for making 2D games in its current state. To summarize my gripes: There is little to no control over the import settings of textures; the actor editor is severely lacking in tools to make larger, more detailed components or debug on-the-fly; The commenting system is frustrating to organize; Paper flipbooks (animated sprites) are limited in capabilities; AI features on the 2D platform are underdeveloped and lack sufficient documentation; features in the project settings that should be enabled by default aren’t and demands a lot of hunting/debugging; and finally default lighting features are not very useful or performance-friendly, especially on mobile platforms. In this 18-page write-up I didn't quite cover absolutely everything that I have included in my game. There tons of smaller cosmetic stuff hanging around the world that use their own intricate systems, but I felt like covering the aspects of my game that were the trickiest and summarized just how much I was able to pull off in such a short amount of time. I intend to continue working on this game's core ideas, but probably not in a 2D platformer, since I mostly got into unreal thanks to it's first-person shooter capabilities. Overall the making of this game was a great learning experience, and I am only excited to expand my abilities with Unreal Engine now that I have tested the waters much more.