

Megaman ZX Demo Report

CS427 – Mini project 1

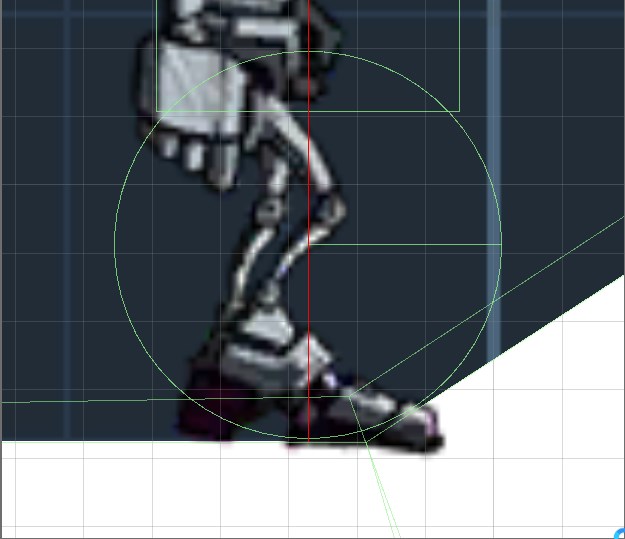
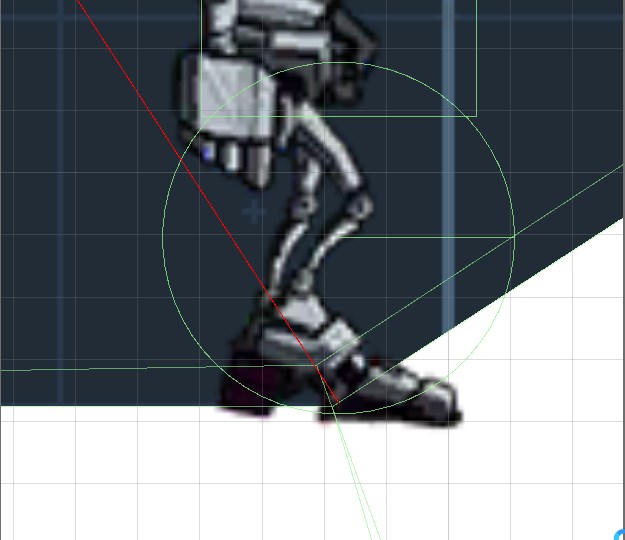
Nguyễn Bùi Lê Vinh | 1651070 | 16CTT

# Why I decided to make a Megaman ZX remake

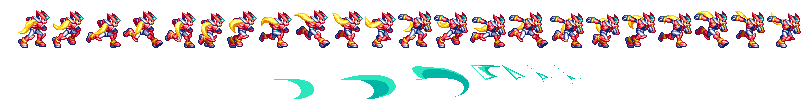
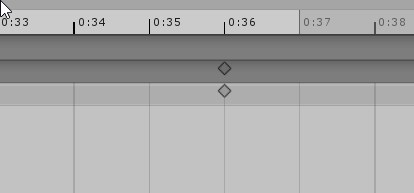
Megaman ZX is an action-platforming game released for the Nintendo DS in 2006. The game was a commercial success from a long running franchise of Megaman games. I chose it because of nostalgia, the assets are available and easily found(albeit not as complete as I hoped), and the fact I have something to directly reference to while making the demo.

# Building a platformer that feels “good”

# Physics:

* As we all know, Unity comes packaged with its own Physics. PhysX for 3D games, and Box2D for 2D games. I prefer the nickname “Blackbox” for Box2D, because we have no idea what the engine is doing.
* Box2D and by extension, Rigidbodies:
  + Pros:
    - Easy to get a simple game up and running with realistic physics.
    - Natively supported by Unity
  + Cons:
    - We have very little control of what the physics engine do
    - **It’s realistic**
* Why is being realistic a cons? It’s due the fact a platformer game is by nature, is **not** realistic. A game character can walk up a slope and stand there, he doesn’t slide down the slope. A game character can instantly go from standing still to running at full speed at a moment notice. A character can jump into the air and then suddenly boost in another direction, etc….Having a realistic physics system in a platforming game makes the controls feels floaty, slippery compared to the snappy, robust controls of old, retro platformers. To make a control system that capture the same snappy feel of those games using Unity’s physics means you will constantly be fighting against the engine itself, a fight where your enemy dictates everything.
* How about we compromise? Let the physics engine do its thing, and only handle specific case that we want?
  + I created a short demo with Unity standard assets, where the player character uses Dynamic mode of the Rigidbody when running on flat terrain, and Kinematic mode when traversing slopes.
  + Kinematic mode is triggered by having a trigger collider along the length of the slope, and upon exiting, Dynamic mode is reactivated.
  + The aim of the test is to reproduce the behavior of classic platformer characters on slope. The character needs to stay still with no input, being able to immediately jump forward or backward, and always stay on the slope unless a jump is commanded
  + The test is included in the folder as Rigidbody Test.mp4.
  + To get the slope angle for traversing, I use a ray casted from the character’s transform position directly downwards. Unity’s Raycast will return to us the distance between the origin the ray is casted from the the collider surface it came into contact with, the surface’s normal vector, along with the collider’s information. From which, we can easily calculate how to convert our character velocity into vertical and horizontal velocity for traversing up the slope.
  + There’s a problem:
    - At the position on the left picture, the player’s feet, represented by the circle collider, should’ve started going up the slope. But because the ray casted from the character’s transform is still on a flat surface, it thinks there’s no slopes yet. And thus the character keeps moving forward, clipping into the ground. This creates problems when the player tries to jump, because Unity will first try to move the character out of this clipping state first, before actually allowing the character to jump on the next press of the button. Or it might just let the player jump immediately. This creates a very inconsistent behavior, especially since upon landing, the player may once again clip into the slope at random amounts.
    - Physics2D.OverlapCircle might be able to help us here, however it will run into problems when dealing with a V shaped slope, or a slope suddenly changing angles
* Thus, I decided to create my own physics system using raycasting to detect colliders and handle collisions myself. This would allow great flexibility to implement whatever kind of behavior I wanted. The only problem is a lot of coding will be needed.
* A raycasting physics system essentially surrounds the character collider with rays, similar to a paramecium’s cilia, constantly checking its surroundings for distance to the nearest thing character will collide with, and if a collision happens(distance = 0), velocity in that direction = 0.
* However, the one I created, despite capturing the snappy and smooth feeling of a good platformer game, is still buggy and does have strange behaviors when the player jump and land on a slope, facing towards the direction going up the slope. Demonstrated in Raycast Test.mp4
* Hence, I used Prime31’s Character Controller 2D, which uses the same principle, but definitely a lot less buggy than mine.

# Animations

* First, we need some context about the NintendoDS limitations and how it affected the way Capcom engineers created the sprites for Megaman ZX.
* “ On the Nintendo DS, we can have up to 128 sprites. Only 32 of those sprites can be affine transformed (rotated, scaled, skewed, etc.). We are also only allowed to address 1024 different tiles to comprise all of our sprites, per engine. Sprites can be made of tiles that use 16 colors or of ones that use 256 colors. Tiles that use 256 colors are twice as large as tiles that use only 16 colors. Another advantage of using 16-color sprites is the ability to use 16 different palettes. When using 256-color sprites, each sprite (and tile) must use the same palette. When using 16-color sprites, we can have one sprite use one 16-color palette and another sprite use a different 16-color palette, even though both use the same tile data. Games often use this trick to make different colored enemies that look the same. The sprites still use the same tile data, but they use different palettes.” – Chapter 6, Introduction to NintendoDS Programming, Jaeden Amero, 2008
* For some clarification, this means the system can display 128 different sprites with different attributes at the same time.
* However, as we can see, it is very limiting. Despite being able to run at 60 fps, the console’s screen resolution is definitely anything but high, and with all the other things that’s on the screen, making the player character animation looks good, feels smooth while not taking up a lot of space is no easy task.
* This is very well shown in the animation for the attack while running. Below are the sprites for running and run attacking.
* Running:
* Run Attacking:
* The Run Attacks is split into 2 parts, the windup(the character prepares to slash), and the actual slash. To make the animation look smooth, for every run sprites, there will be a matching windup sprite, and a matching slash sprite.
* The sword is separated into its own sprites for this animation so that instead of needing, 55 sprites for the slash, only 17 is needed. Very impressive, Capcom engineers!
* How this animation works: When the player is running, the run animation loops normally. Upon attack input, the game get the current sprite [i] in the run loop array, fetch the windup sprite [i+1], then the slash will be from slash [i+2] to [i+4], and finally return to the run loop at run loop sprite [i+5].
* This, however, is a disaster for Unity’s Mecanim animation system, also known as the animator.
  + The animator transition system upon switching to another state, will always start at the same point, no matter where the previous animation ended.
  + It doesn’t really know where it is throughout the animation, normalized time value is very inconsistent, and is very frame rate dependent. The engine when not able to process fast enough will
  + skip a few frames of the animation to catch up, this is usually not noticeable by the human eye
* This has been a long running problem for a lot of Unity game developers, until StateMachineBehaviors, a feature not well known to many, is introduced. This allows us to repurpose the animator into finite state machines(to make AI for our games) and in this case, achieve what we needed, transition between states with a dynamic starting point. The 3 scripts for these 3 states can be found in Assets\Script\Animation, RunAtkWindup for transitioning from Run -> Windup, RunAtkSlash for transitioning from Windup -> Slash, and RunReturn for transitioning from Slash -> Run.
* Using PlayinFixedTime, we can start a state a any time throughout its animation, however, this function is extremely badly documented by Unity in my opinion.:
  + public void **PlayInFixedTime**(string **stateName**, int **layer** = -1, float **fixedTime** = float.NegativeInfinity);
  + “fixed time” - Time offset (in seconds)
  + Unity doesn’t clarify what “seconds” does it mean here. The animation screen “seconds”, or our real world seconds.
  + In my opinion, the animation screen “seconds” is a pointless, misleading and confusing measurement that serves no purpose and I strongly recommend everyone to click on the gear button on the top right of the animation screen and select Frames instead.
  + Current time to be used for calculating the input value for fixed time can be get from float time = animator.GetCurrentAnimatorClipInfo(0)[0].clip.length \* Mathf.Repeat(stateInfo.normalizedTime, 1);
  + Offset to decide when to switch states can be calculated with (Number of frames to offset) / Sample Rate. For example, if we want to play an animation for 4 frames at a sample rate of 60, we switch state at timestart + 4/60
* However, the problem with Unity skipping frames is definitely much more accentuated with this method of transitioning between states, shown in Unity Frame skipping.mp4 where sometimes, for example, despite the length of the animation reported by unity is 0.79 seconds, Unity can suddenly go from time = 0.62s, to 0.68s and suddenly goes straight back to 0.05s, skipping the frames from 0.69s to 0.79s. This means the manual transition code never gets executed, and the player gets stuck in the animation until Unity decides to properly run an animation to the end.
* This is only somewhat fixed by me setting where to loop back earlier than the actual length of the animation by a healthy amount. A few frames…or sprite might be skipped or run longer than it should, but at least, Megaman isn’t stuck dabbing in the middle of a fight!

# Sound effects.

* For an unknown reason, the sound effects and voices ripped from the NDS game is higher pitched than how it sounded on the game console. If played at pitch 0.9 (10% less pitch than the source file), everything sounded normal again.
* Music from ripped original game cannot be used, as it’s made from playing different short sound clips of different instruments, mixed together, reminiscence of how music is played in the first generations of game consoles.
* The sound effect for wall sliding needed a timer to delay the playing of subsequent sounds in its loop, if played till the end, it wont sound like it should. If played on Update() without a delay, it becomes a screeching noise that hurts the listener ears. Sound file is included in Report folder.

# Visual effects(Particles system)

* The shuriken particle system of Unity(the newer one, VFX Graph, is only available for HDRP) gets the job done, albeit, it has some strange limitations:
  + When an enemy explode, its head and 2 feet flies off in different direction, there are 3 different sprites for this, but the only way to implement this is to make 3 particle systems, each with a sprite renderer material, texture animation ticked, using grid animation with a fixed row. If sprite is selected, it instead shoots off 3 particles with each changing from head -> feet 1 -> feet 2

# Other Unity things:

* If a GameObject has many children, upon selecting, the transform shown in the Editor(Inspector) is the average transform of the parent and all the children. However, when scripting, if the script is for the parent GameObject, it will use the parent GameObject transform for the calculations instead of the average transform shown to us. This has created a strange bug where the raycasted from the enemy transform position to detect the player actually originated from his feet instead of his center point(the transform shown to me), which took quite some time to debug.
* If an Input.Getbuttondown is used in FixedUpdate(), it may return true multiple times per button pressed. Definitely put input related codes into Update() instead.
* Project Settings->Script Execution Order editing is a must if we’re building our own physics, to ensure collision detection is always done before any movement calculations is done.
* Unity doesn’t handle accurate Timing well at all. Time.time will return values all over the place.
* Animation Events are a great way to sync things with animations, for example, foot step sounds only played when the sprite feet stomps on the ground. Beware of frame drops.

# References:

1. <https://github.com/prime31/CharacterController2D> <https://www.youtube.com/watch?v=MbWK8bCAU2w&list=PLFt_AvWsXl0f0hqURlhyIoAabKPgRsqjz> - Playlist for making a robust 2D platformer using raycasting for physics
2. <https://roystanross.wordpress.com/2015/05/10/custom-character-controller-in-unity-part-6-ground-detection/>
3. <https://www.gamasutra.com/blogs/YoannPignole/20131010/202080/The_hobbyist_coder_1_2D_platformer_controller.php?print=1>
4. https://patater.com/files/projects/manual/manual.html#id2614152