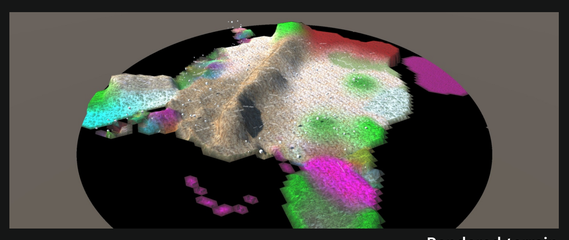
NODE BASED RELATIVE TERRAIN RENDERING SOLUTION USING UNITY3D

WHAT IS THIS?

A custom 3D open world game world loading, managing and rendering solution. Its made in the Unity3D game engine but the project itself is very similar creating a game engine of my own.

WHO MAKES THIS?

I'm Kristo Männa and I make this. I want to make my own Open World RPG and creating this world system is one stepping stone along that path. I have been developing this project for 6 months with around 500 hours having been sacrificed for its development.



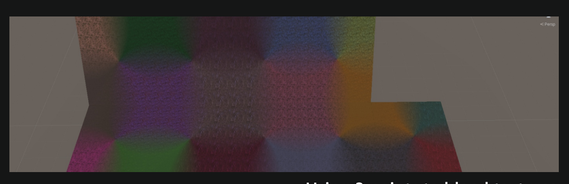
Rendered terrain

PROBLEM

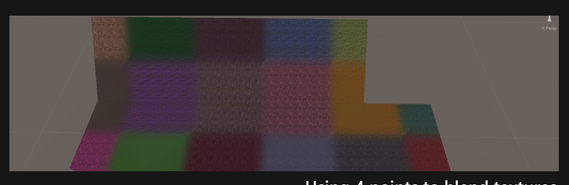
The problem is that open world games worlds are big and loading the whole world in at once, if possible, is too slow. There are two widely accepted general solutions for this: The first one being to subdivide your world into levels and have only one level loaded at a time. Classic games such as Mario use this approach. The second way is to subdivide your world into chunks which are a collection of world data tied to a specific world coordinate. So then what you can do is only load in the chunks that are close enough to the players coordinate. Virtually all open world games use a variation of this. Both of these solutions work well, but they each have drawbacks. While level based world system are very easy to create and manage they require some sort of loading transition to go from one level to another and that is not very open world like. The problem with a chunk based solution is that to get from A to B naturally you have to cross the chunks between A and B and that while being very realistic isn't always fun. As a game designer I would like to have the freedom of the level based world system and to have the seamlesness of the chunk based world system without using any loading screens.

SOLUTION

My proposed solution to this is basically have chunk based worlds but instead of the chunks having absolute coordinates their transforms are defined by their relations to other chunks. And I call them nodes not chunks. So basically it is a node graph. So in order to load in nodes you have to convert their relative transforms to the world transforms. This way the game designer can connect any nodes they want to create their maps. But this solution is not without its drawbacks with the main one being that it is a custom solution and you have to implement it yourself. For most cases implementing this system will be overkill but if you want to have a lot of freedom in designing your open world game or dislike loading screens as much as I do this system can be considered.



Using 3 points to blend textures



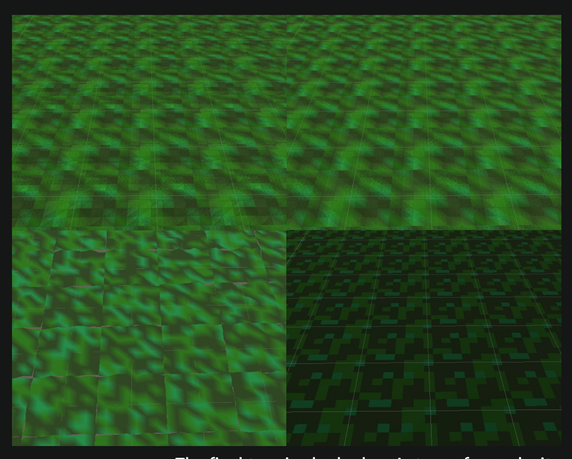
Using 4 points to blend textures

PLANNING

Currently i am on my third iteration of this project because on the previous two iterations I had managed to code myself into a corner. To remedy this I decided to give throw planning a shot. I wanted to avoid code design that gets more difficult to manage the longer you touch it. I knew I needed to plan out the whole project. This meant that the planning method I was going to use had to be very simple, so I wouldn't drown in complexity yet powerful enough to get my ideas across. Luckily there were only a few things I cared for in the plan: classes — their state and desired functionality, how they are related to each other I also wanted to show when a class was abstract. So I came up with a simple planning system of boxes and arrows. Boxes are classes, classes have state and functionality and there are arrows to show how the classes are related to each other. Abstract classes are marked red while normal ones are black.

RENDERING TERRAIN

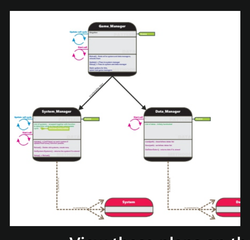
So each node can have some terraforming data. When the nodes are loaded that terraforming data is blended with other nodes terraforming data and converted into a grid of terrain cells. Each cell has a bunch of values like height, color, texture and so forth. Then these cells are converted into planes where a plane is a plane mesh tied to 8x8 cells so that each vertex corresponds to a cell. Then the planes are rendered using instanced rendering. Instanced rendering is a way of batching together draw calls that use the same mesh and this can give massive performance gains over using a single draw call foreach object. When it came to rendering the planes I ran into a lot of trouble.Because I am using Unity's surface shader I can't use the geometry shader step sothis mean I have to sample the cells per fragment instead of per vertex.My biggest mistake was to make the shader do too much at Foreach fragment I wanted the shader to blend nearby cell values, use triplanar shading, and have a bunch of cool things like emission and a height map. The way I had done it meant that foreach fragment I then had to do 4*4*3*3 = 144 texture reads to combine all the texture values together and this wasn't exactly fast. So I decided to scrap most of the features and make a fast and stylized terrain rendering system. In the end I reduced the texture reads down to 6 per vertex — 3 cells and 2 textures per cell for diffuse and normal.



The final terrain shader has 4 steps of complexity

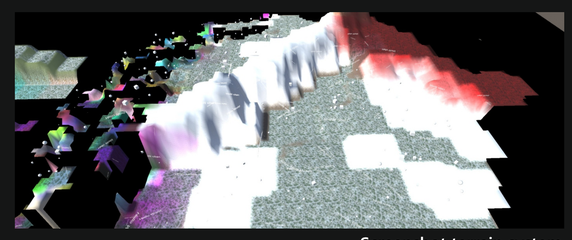
SYSTEMS

To implement the world system I use the following systems: The loader system converts nodes from local space to world space and loads in range nodes around the player. The builder system creates pooled Unity's GameObjects for each loaded nodes GameObject data. The Terraform system creates and renders planes generated from loaded nodes terrain data. The Dynamer makes it possible to have GameObjects that can move between nodes and still be serialized and deserialized at their new locations. The Instancer renders instanced objects tied to loaded nodes for example grass tuts, plants and imposters. An impostor in game architecture is a picture system of an object. When the object is far away instead of rendering it you render its impostors picture and this sometimes even gives pixel perfect results while being cheaper to render then the object itself. The Pathfinder manages pathfinding across and inside nodes. The Occluder manages a bunch of voxels and foreach of those voxels it calculates an IsVisible value if the voxel is visible or not to the camera. The Editor is an extension of the Unity editor using a editor window as a base and creating a custom window inside of it. To add a bit of style to the editor I used Unity's still in the making UIElements system which makes it possible to create editor UI similarly to how you would create a website.



View these plans on the project website

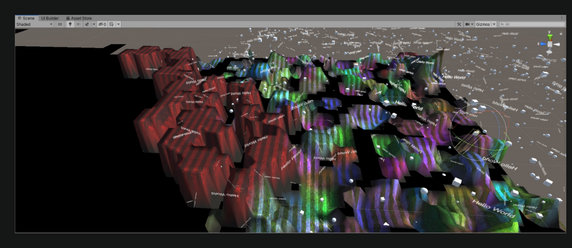
Editor window



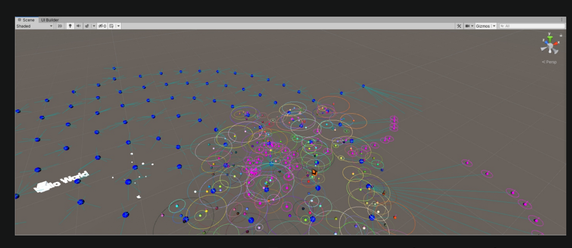
Screenshot terrain system

MULTITHREADING

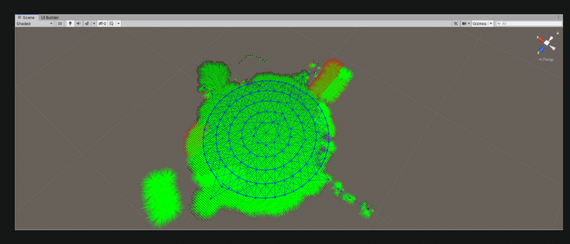
Another thing that frequently gets me into the habit of bad code is coding for performance. I decided early on that this project was going to be multi threaded. This would allow me to keep my code simple because there would be enough performance to go around. So how my threading system works is that you have three parts to it. Part one is the thread pool, it does queued work max once each frame, this is ensured by making the worker threads wait for the main thread to open a barrier when they have finished their work. The second part is the systems which have two states of execution — working and syncing. Working state is done on subthreads and syncing state on the main thread. The state switches once all systems that influence each other finish their current state. The third part is what this allows me to do — on the working state all systems can read the global data of other systems and on the sync state they write their local state to their global state. So this allows me to really easily create systems that can access each other's data while being run on multiple threads simultaneously.



Terrain system output



Terrain system node terrain data view



Green lines drawn between nodes and their terrain cells

WHAT'S LEFT

The project is not yet fully completed — the Instancer, Occluder and Pathfinder are still yet to be added. I expect the Occluder to put up a bit of a fight. To see what I have made so far check out the materials on the website that is linked with my student project competition entry. Also you can see the plans for my systems there.

CONCLUSION

This project has been a wild ride with the rendering of the terrain planes and creating the editor system taking up way more resources than I expected but that just made it so much better when I did finish them. If you were somewhat inspired by this poster and want to create your own world thing using the Unity game engine then I would strongly suggest that you firstly plan out your project, secondly not underestimate how much time making a custom editor solution is going to take and thirdly remember that no matter what you do writing shaders will cause unexpected bugs, and they will be a pain in the ass to solve so block out time for that beforehand

KRSITO MÄNNA

Computer Science BSc

Faculty of Science and Technology

Institute of Computer Science

UNIVERSITAS TARTUENSIS 1632

IT Akadeemia