Approach to Procedural Environment Generation

Project Report

Presented to Dr. Clark Verbrugge

COMP 521 – Modern Computer Games

By

Mathieu Boucher 260589672

Yanis Hattab 260535922

April 14, 2015

McGill University

Introduction

In recent years, modern video games have improved exponentially in terms of the size of the world they offer and its fidelity to reality. Games like Assassin's Creed Black Flag or Just Cause 2 offer thousands of kilometers of environment to explore and its realism is sometimes striking. Such virtual worlds are expensive to make and are content heavy, we aim to ease such projects by building a framework that automatically generates realistic environments in a procedural way while maintaining high fidelity and flexibility for the game designers. We want to be able to generate a 3D terrain randomly and populate it with realistically behaving agents (animals for now) that would use steering behaviours to display improvisational and natural movements.

Algorithms to produce realistic terrain in a procedural context are useful for many applications beside video games and many of theses algorithms generate terrain without any user-input or interaction to customize it, while the terrain produced by the latter mean may be realistic, the designer is not able to specify certain key features in the generated terrain. In our project, we explored a way for the user to be able to seed a height map to set key points in the terrain instead of relying on a semi-random seeded height map (though we also cover this). This allows the user to create terrain with specific properties. The terrain generation implementation and the report sections on it are mainly done by Mathieu Boucher although it is happening in a spirit of collaboration.

Steering behaviours have been around in the video game industry for a while and represent a simple and efficient way to produce emergent behaviour and have non-deterministic AI. In our case we want to aggregate the different steering behaviours in one single base class that will be extended by agents so as to have the game designer only think of higher order goals for his agents and leave the locomotion primitives to the base class. The steering behaviours' implementation and report sections are done by Yanis Hattab with their inclusion in the terrain done and tested cooperatively.

Terrain generation background:

As games increase in complexity and scope, player demand for content, such as terrain, is heightened. Moreover, in certain styles/genres of games, terrain is an important asset (Forbus [1]). In the past, every component in the game was hand-crafted by developers, requiring a non-trivial investment of resources to accomplish. This led to the idea of automating the generation of these components (procedural content generation) (Nelson and Mateas [2]). Procedural content generation has the advantage of lessening the load on the developers, thus reducing resources required to generate a given asset. According to other procedural content generators ([3] J. Doran), five aspects are important for any procedurally generated component:

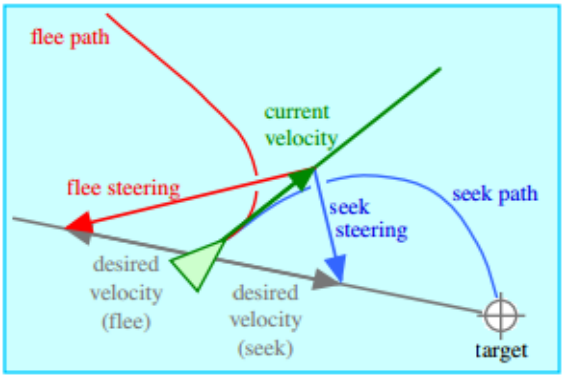
« • Novelty: contains an element of randomness and unpredictability.
• Structure: is not merely random noise, but contains larger structures.
• Interest: has a combination of randomness and structure that players find engaging.
• Speed: can be quickly generated.
• Controllability: can be generated according to a set of natural designer-centric parameters »

In our project, we concentrated our efforts on the novelty, structure, speed and controllability aspect of procedurally generated terrain. Interest,without a good sample size, is a hard qualitative data to measure without adequate resources, so we did not factor it in our qualitative analysis of the procedurally generated terrain.

Similarly, the controllability aspect was neglected in our project as it would require the abstraction of our terrain generation algorithms with an implemented GUI that an artist could you use to create and modify the generated terrain for specific styles and genres of games.

Steering behaviour background

Steering behaviour is an approach to designing AI that aims to produce autonomous agents (either in a simulation or in video games) that behave in a realistic way and display patterns of emergent behaviour. The core supporting the implementation of steering behaviours is to use an Euler Integration method to move the agents and then apply a certain number of forces to the velocity of the object to produce a wide variety of desired behaviours. The emergent behaviour comes from the combination of those forces which may not always produce the same movement as well as a possibility to introduce randomness in the intensity and direction of the



forces. Thus there is a part of improvisation in their way to move around, something that would be harder to obtain using algorithmic static pathfinding. The main steering forces discussed in Craig Reynolds' paper are the *seek* force that pushes an agent towards a target, a *flee* force from that target, and a *wander* force that moves an agent randomly but in a continuous fashion. On top of that, composite behaviours can be built like *pursuit* or *evade* another agent, collision avoidance (flee obstacles), path following or leader following.

Another useful application of the steering behaviours approach is to produce flocking behaviours that greatly resemble their natural model. Using this approach we can produce flocks of extremely large number of agents in an efficient and rather simple way that consists of the combination of enforcing simple rules, the whole process remaining very performant and uses low resources. The rules needed to implement flocking behaviour are:

- Alignment : An agent should move in the general (average) direction that its neighbors are moving

- Cohesion : An agent should stay close to the "Center of Mass" that is the averaged position of neighboring agents

- Separation An agent should maintain a certain distance from its neighbors

Steering Behaviours' Methodology

Regarding steering behaviours, our initial approach was to produce a script for each of the basic behaviours and have one object in Unity be controlled by that script in order to test and debug the behaviour. Our object needed a few fields to allow us to control its motion through forces: A speed, a mass, a velocity, a maximal velocity and a *currentSteering* vector that would be the cumulative forces of steering we will apply to the velocity. The *seek* and *flee* behaviours were relatively easy to implement after we defined and found optimal values for the influence factors we would give to the steering produced, taking the steering vector and adding it to the velocity directly would lead to sudden movements that were unnatural. We added public Transform fields to the objects to store a *target* and an *enemy*, the first one to seek and the other to flee. After trial and error we found that for a sphere of scale 1, taking 0.2 of the computed steering vector often gave the most realistic results with an agent that changed direction fast enough as to not miss its target but slowly enough to remain realistic. To increase the realistic appearance, we added an *arrival* to the *seek* behaviour, whereas the agent increasingly slows its speed as it nears the target.

wander:

That behaviour is meant to guide an agent in a random like wandering movement, we do so by picking a point along the velocity vector of the agent that is at a specified distance from the agent and call it the circle center. From that point we use Unity's *Random.OnUnitSphere* to get a random Vector3 the we make that vector start at the circle center to produce a randomly generated displacement that can be applied to the velocity to slight change the agent's heading.

followLeader:

This behaviour builds upon the *seek* behaviour, here we specify a public Unity object (another agent) that is the leader. From the leader's velocity we compute a position along that vector that is behind the leader's position, we have a parameter that indicates the distance at which an agent should follow his leader. We then return a steering force to seek towards that position.

separation:

This one is needed as when many agents follow one single leader, they display flocking behaviour and we must prevent them from colliding with each other. To do so we use Unity's *Physics.OverlapSphere* to get all colliders with a certain distance of the agent (this passed as an argument calle*d personalSpace*) and check if they have a tag "Follower" if so we return a combined force to flee from all close agents. The issue with combining the force is that sometimes it cancels out as the agent is repelled in a symmetric fashion, to fix that we added rigidbodies to the followers so that they can't collide.

followPath

Using *seek* and a provided array of Vector3 points, this uses the class field *pathIndex* to remember which point of the array is the next destination and if the agent's distance to that point is smaller than a provided threshold, the following point in the array is set as the current target. If we reach the end of the array we start back from the beginning.

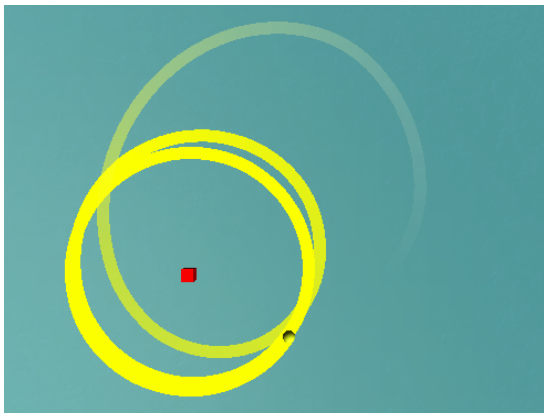
After testing and debugging the different behaviours, we decided to incorporate them all in one base class called Steerable. This is aimed towards the goal of ending up with a framework that game designers could use to generate realistic environments with animals, thus any agent desiring steering behaviours could extend that and then use its protected methods and fields as applied to its very own context. This gives the flexibility to the end user to incorporate his own custom goals and planning strategies as well as custom heuristics. The class implements a method for each steering behaviour and takes according arguments and it returns a Vector3 that is the steering force obtained. The class also contains helper methods that can be used to constrain our agent in our randomly generated terrain such as *findTerrainHeight()* that returns the height (y value) of the terrain at the current x and z position of the agent or *avoidTerrain()* that constraints an agent to a valley by returning a fleeing force if the agent gets close to the valley borders. The method *findTerrainHeight()* uses Unity's Raycast to look below or above the agent to find a collider with a tag "Terrain" and if so it returns the y position of the hit point as the terrain height. The *avoidTerrain()* method works similarly and Raycasts towards the agent's velocity to find a collider tagged "Terrain" and avoid collision with it by fleeing it.

Finally we needed to create examples of fauna that would use the Steering class, for demonstration purpose. Because we generate an island that is surrounded by sea, we decided to have a bird that would fly in the sky in an orbiting way similar to a seagull, a school of fishes that would swim together in a pound or in the sea and a goat that would wander on land. These are just examples of animals one can build using the steering behaviour we implemented, game designers can come up with more complex fauna as they combine the different steering behaviours. The seagull uses *seekAndOrbit* to fly in circles and when the player gets close to it, it would start following the player around, that can be used as a game feature where you need to lead birds in specific areas. The fishes follow their leader that is wandering around and they stay in the water using *avoidTerrain*, if the player gets too close to them, they *flee* away from him. The goat is wandering on the land and it follows the terrain using findTerrainHeight, thus it can climb mountains but it will keep out of water by maintaining it fleeing terrain with height less than 0..

Steering behaviours' Results

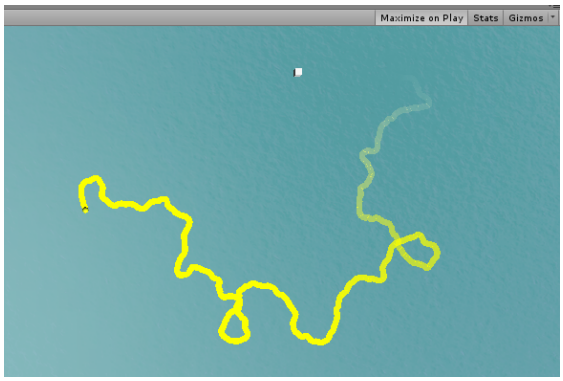
To be able to represent the trajectory taken by agents as they are influenced by steering behaviours, we attached a trail renderer component to the object and took screenshots of the trajectory painted by the trail.

In order to implement a realistic seagull behaviour, we tweaked the traditional seek behaviour to obtain *seekAndOrbit()* which keeps more of the original velocity and applies very little steering in order to obtain an orbiting behaviour. In nature we can often see birds circling around sources of food, in this case we have seagull circling in the air near the beach above potential fish underwater:

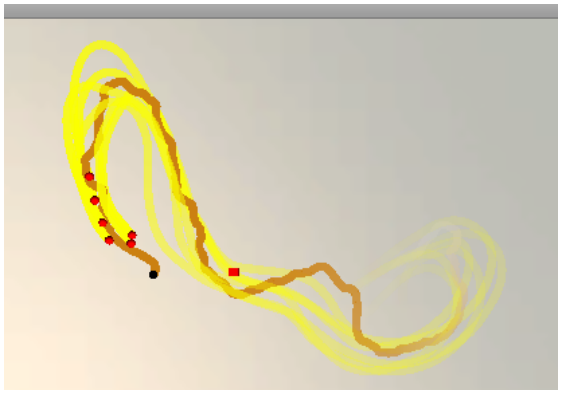


Yellow sphere seeking and orbiting around red cube

The wander behaviour produces wildly different behaviour just by changing the influence value of the steering produced by the *wander()* function, here one can see the trajectory taken by a wandering agent with an influence proportion of 0.1 compared to an influence of 0.1, we can see how much more randomly the agent wanders around with an influence factor of 0.9:



The follow leader behaviour produces satisfying results when combined with the *separation* method which ensures that followers are not colliding so much the group behaviours looks natural and members of the following group don't stay too much behind or go in front of the leader. The separation is one of the flocking principles, the alignment and cohesion being provided here by the fact that all followers chase the leader and have the same speeds. To achieve perfect non-collision we added a rigidbody to the following agents so that Unity handles their collisions. Here is a screenshot of 5 red spheres following a black one, one can see that not all followers take the same path which leads to a natural looking result:



Creating the birds, goats and fishes based on the Steerable class was quite trivial afterwards and we had the ability to give them realistic and flexible behaviours.
The birds circled around a point (Vector3) and if the airplane (player) came close enough to them they would start following it until he gets too far (if its speed is greater).
The goat remains on the ground and if its height goes below 0 it changes direction as to not go in the water (water level is at y = 0). The goat also followed the terrain height using the *findTerrainHeight* utility function.
The fishes were spawned as fish schools that had one fish as the leader, that wandered around, and the other fishes following that fish using *followLeader*. The fishes used *avoidTerrain* to stay within the water and not enter the terrain.

To combine all those animals, we have a script Populator that has access to the height map of the generated terrain and spawns fish schools, bird flocks or goats on and around the terrain. You can specify the number of each animal and it has heuristics as to be sure to spawn goats on land and fishes in the sea or in ponds. Birds are spawned flying around peaks (mountains). This approach fills the environment with realistic animals that don't crowd an area and are evenly spread throughout the terrain, also giving control on how much animals are spawned.

Terrain generation's Methodology :

We used an iterative 4-step approach to build our project. That is, we first set an objective to attain in our respective tasks and we then go on to code it. After testing the code and fixing any occurring bugs, we then repeat the process. For the procedural terrain generation component, the objectives were:

1. Implement a noise generating algorithm to generate a height map
2. Map the height map onto a mesh (Unity component) and render it
3. Create a custom texture for each mesh that maps different vertex heights to different textures
4. Add progressive granularization feature to meshes with respect to a moving target (unfinished)
5. Implement a variation for the diamond square algorithm that calculates a height map with a border completely seeded to allow extensions to current terrain (unfinished)

We decided to implement the diamond-square algorithm as our noise function because it allowed easier seeding potential than perlin noise, thus allowing a better overall controllability measure aspect to the procedural terrain generation. Indeed, after seeding heights at points spaced evenly apart, the algorithm will interpolate between these points and add a certain novelty to the terrain, one of the five key components to procedural content generation. By allowing seeding to the user, the latter may also define to some degree the structure of the terrain.

The diamond square algorithm first requires a seeded height map with seeds spaced evenly apart. It then goes on to generate the square heights (center vertex of a given square) between the seeds by averaging the four heights of the four corners of the square plus a random amount proportional to the distance between the points (see Figure 1) . Afterwards, it computes the diamond heights for each square generated above, which are the four midpoints between the original seeds, by averaging the square height with the seed heights and adding a random amount proportional to the distance between the points (see Figure 2). Note that both square and diamonds heights in the images below rely on a wrapping behaviour to compute their heights. For example, in Figure 1, for square height calculation 2, the upper right corner would be the (0,0) square in the image. After a first round of calculations, we reduce the scale which increases the granularity of the terrain. As a consequence it reduces the random amount that can be added to the square and diamond heights at each of the iterations. The algorithm stops once the scale is less than 2, implying every entry in the array has been assigned a value.

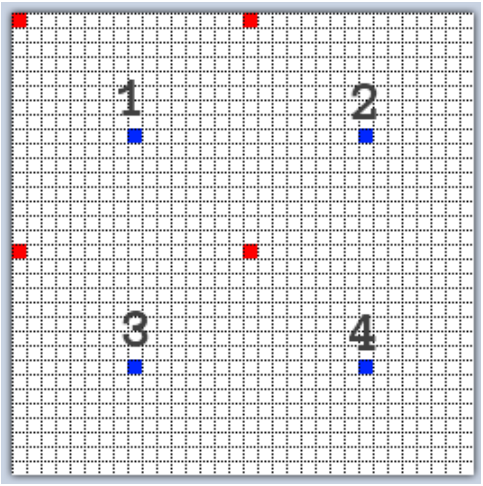


Figure 1 : Square Heights

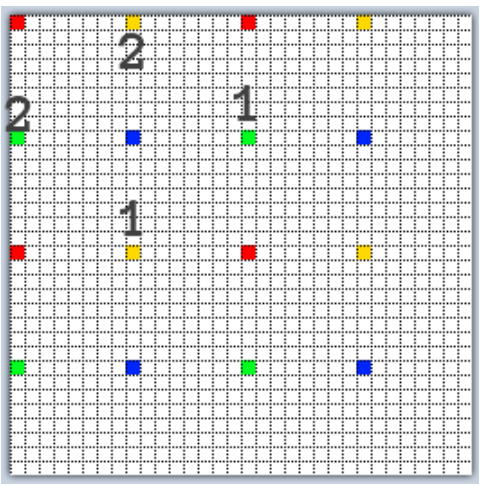


Figure 2 : Diamond Heights

To address the wrapping behaviour, we decided that we only average the values that exist around a given point, effectively eliminating wrap around behaviour.. That is, if we want to calculate a point's height and only two of the heights exist in the array, than we average only these two points.

After mapping the height map values to a Unity mesh and placing the heights one unit (meter) apart, we realized that for a realistic island with at least a kilometer wide by a kilometer long, we would require over 1 000 000 vertices for mesh. Unfortunately the unity mesh API does not allow meshes with over 65 000 vertices. To circumvent this, we decided to split the huge mesh into many sub-meshes with the size being specified by the user, thus increasing controllability of the terrain being generated. We then went on to generate random islands with areas of at least 1 km^2 but the huge number of vertices made the generation long and time-consuming. Rendering was also problematic, since the number of frames per second was in the 15-20 range, hindering the simulation's playability. However, we noticed that the spacing of vertices (1 meter or unit apart) did not necessarily give a visible granular effect to terrain when compared to a relaxed spacing of 5 units. Following this, we increased the controllability of the terrain generation by allowing the user to specify vertex spacing to create larger, coarser worlds and doing so, allowed rendering and time-generation to improve, thus ameliorating the speed aspect of terrain generation.

For each mesh generated above, we created a custom texture based on the height of the vertices: low height has sand, middle height has grass etc… The textures and thresholds of which are all specified by the user, thus allowing greater controllability. To generate our textures, we first tried that each vertex has a 16x16 random pixel sample from the texture specified by the

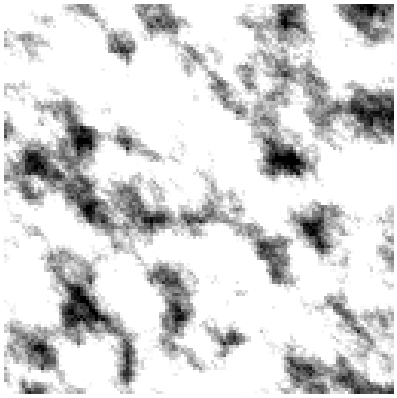
vertex's height. However, this lead to very large textures yielding poor results in terms of generation time and rendering performances. To circumvent this problem, we decided that each mesh would have the same amount of pixels for its texture, the value of which is decided by the user to allow greater controllability, and each vertex would be mapped to a specific pixel on the texture with a single random pixel from the vertex's height texture. By using a reasonable number of pixels for the texture, rendering and time generation greatly improved, yielding favorable results even for large terrain generations. To allow even more texture richness, we implemented a noise algorithm on the heights in a mesh without actually varying the heights of the vertices, but rather computing new values based on the existing heights. This yielded a noised mesh height map that was used to compute the height to pixel mapping for the texture.

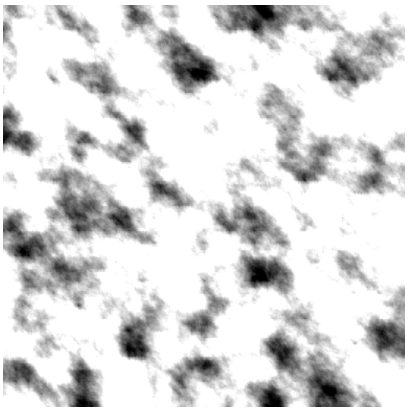
To improve time generation and rendering capabilities, we started to implement a progressive granularization when a target moves closer to a given terrain section. Terrain meshes far away would have been coarser (larger vertex spacing, less triangles to approximate landscape, smaller textures) while closer meshes would have been much less coarser (smaller vertex spacing , more triangles and greater textures). We have reason to believe this would have improved the rendering speed since the reduced number of vertices and texture quality far away would lower the graphic load in the simulation.

To be able to render entire worlds for a moving target at run-time, we would need to be able to connect an existing border mesh heights as seeds to the new height map. The latter procedure would require a new implementation of the diamond square algorithm since seeds are not evenly spaced, but rather border specified. The implementation of such an algorithm proved tricky and we did not have time to finish it. However, if the algorithm were coded, we could have easily expanded our meshes through the use of border sharing and coupled with progressive granularization mentioned above and through the use of Unity coroutines for in game mesh generation as well as mesh deletion when the target is very far away, we could have rendered an entirely playable random world for our simulation.

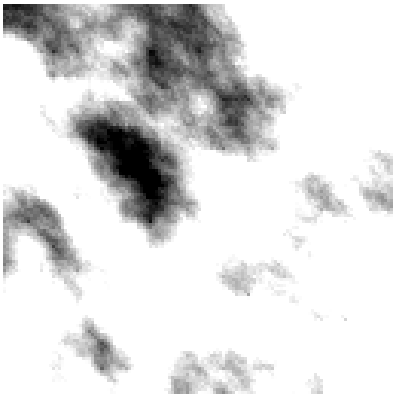
Terrain generation results:

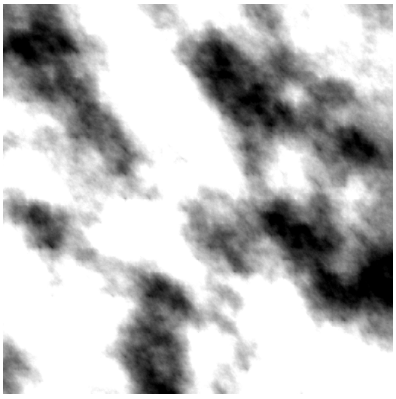
Implementation of the Diamond Square Algorithm :





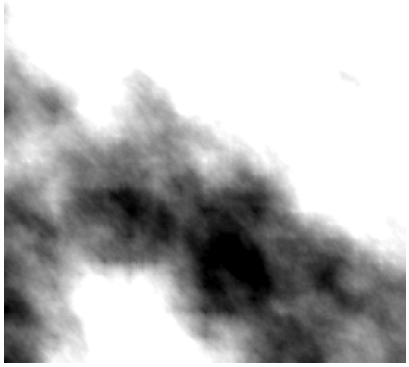
The diamond square algorithm that we implemented yields interesting and rich height maps. Each of the following 5 randomly generated figures are 512x512 pixel textures. The first figure has seeds spaced 16 entries apart, the second 32, the third 64 etc… As we can see from the figures, a good spacing between seeds for a smooth bumpy terrain are 16,32 and 64 or a 32th, 16th or an eighth of the size of the height map. For greater spacings such as 128 and 256, we see that get less smaller structures, but rather a small number of big structures. Therefore we can conclude that for controllability purposes, the seeding spacing is an important parameter and should thus be available to the user to specify its value. We also see that there is no wrapping behaviour on the height maps, as we wanted.

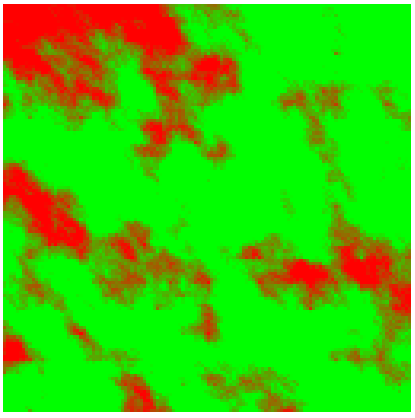


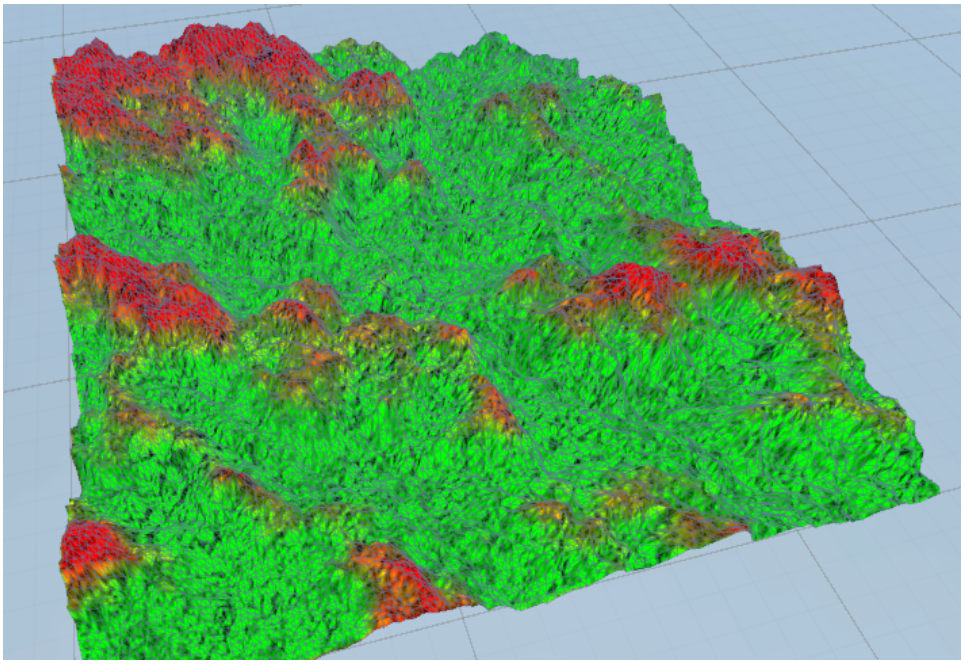


Mapping to mesh:

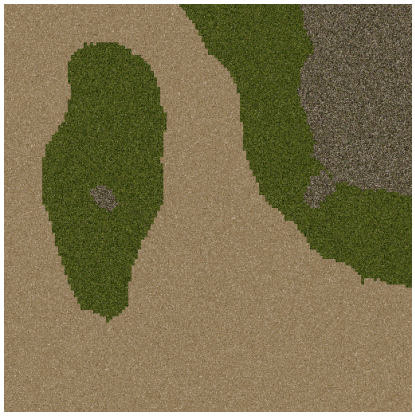
The two figures at the bottom of the page show first a height map and its corresponding mesh transposition for a terrain with vertex spacing at 20, random height change at 7 and mesh length at 128 yielding a terrain of over 6.5 km^2 with rendering time less than two seconds.



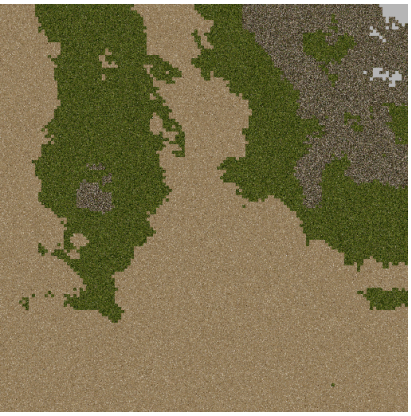
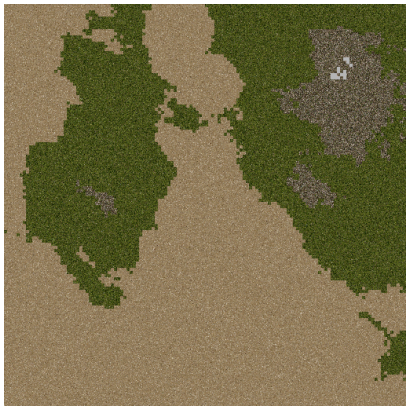
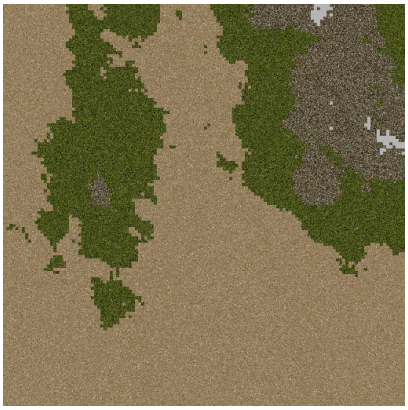




Texture Generation:



The first image on the right yields a given texture from a height map without any noise iterations (diamond square algorithm run). The following four graphs on the bottom show in order from left to right 1 noise iteration, 2 noise iterations, 3 noise iterations and 4 noise iterations with vertex spacing 16 and random displacement 40 on the right texture. As we can observe for the given textures below, adding more noise generations doesn't improve the previous noise iteration, it just creates a new texture that could have been obtained from the original texture. Therefore, a single noise iteration can add richness to textures without compromising computing. To reduce pixelization when close to the mesh, we used a bilinear filter mode to average between texture samples rather than a point mode filter. Moreover, the time required to generate these textures, even with noise, take less than half a second with little impact or performance cost at the rendering stage.


Conclusion

In conclusion, we achieved a nice looking terrain but we were not able to attain all our goals for the procedural terrain generation component of the project, however an interesting way to expand upon our milestone would be to implement the progressive granularization discussed in the terrain methodology section as well as completing the seeded border height map calculations. Other interesting features to add would be implementing initial world sizes in non-powers of two since the current implemented diamond square algorithm for computing height maps relies on this fact, a system to clamp vertex heights to prevent points sticking out of the terrain, efficiently calculating each individual vertex normal to add more depth to textures with respect to lighting, adding a generated normal map to textures to improve its overall quality, reducing pixelization within generated textures and correctly supporting noise generated textures to avoid inconsistencies (e.g noised grass patch under water should not be allowed to be generated). Furthermore, instead of using a height to texture mapping, we could implement Whittaker's biomes through temperature and precipitation tables to texture mapping. Some improvements to the the user interface (through text file currently) for seeding the height map could also be made.

Regarding steering behaviours, our framework is useful and can be extended to implement the wide variety of behaviours that game designers can imagine. The issue though is that combining multiple steering forces ends up being complicated as some forces cancel each other while others produce undesired oscillation. To fix that a possible improvement is to contextualize the forces and have an algorithm to makes sense of them. An idea discussed by Andrew Fray in Context Behaviours Know How To Share is to make a map of the directions in which the agent can go and have steering behaviours fill it, a steering behaviour would then either be an interest (seek coin) or a danger (avoid collision). That would keep the emergent aspect and allow more complex behaviours to be implemented as agents would analyze their environment and react to it, displaying context awareness.

References for Terrain Generation

[1] K. Forbus, J. Mahoney, and K. Dill. How qualitative spatial reasoning can improve strategy game AIs. IEEE Intelligent Systems, 17(4):25–30, 2002.

[2] M. Nelson and M. Mateas. Towards Automated Game Design. Lecture Notes in Computer Science, 4733:626, 2007.

[3] J. Doran and I. Parberry. Controlled Procedural Terrain Generation Using Software Agents, 2010

[4] T. Archer. Procedurally Generating Terrain, date unknown

References for Steering Behaviours

Steering Behaviour, Craig W. Reynolds
http://www.red3d.com/cwr/papers/1999/gdc99steer.pdf

The Three Simple Rules of Flocking Behaviour, Vijay Pemmaraju
http://gamedevelopment.tutsplus.com/tutorials/the-three-simple-rules-of-flocking-behaviors-alignment-cohesion-and-separation--gamedev-3444

Introduction to Steering Behaviours, Juan Belon Perez
http://www.gamasutra.com/blogs/JuanBelonPerez/20140724/221421/Introduction_to_Steering_Behaviours.php

Context Behaviours Know How To Share, Andrew Fray
https://andrewfray.wordpress.com/2013/03/26/context-behaviours-know-how-to-share/