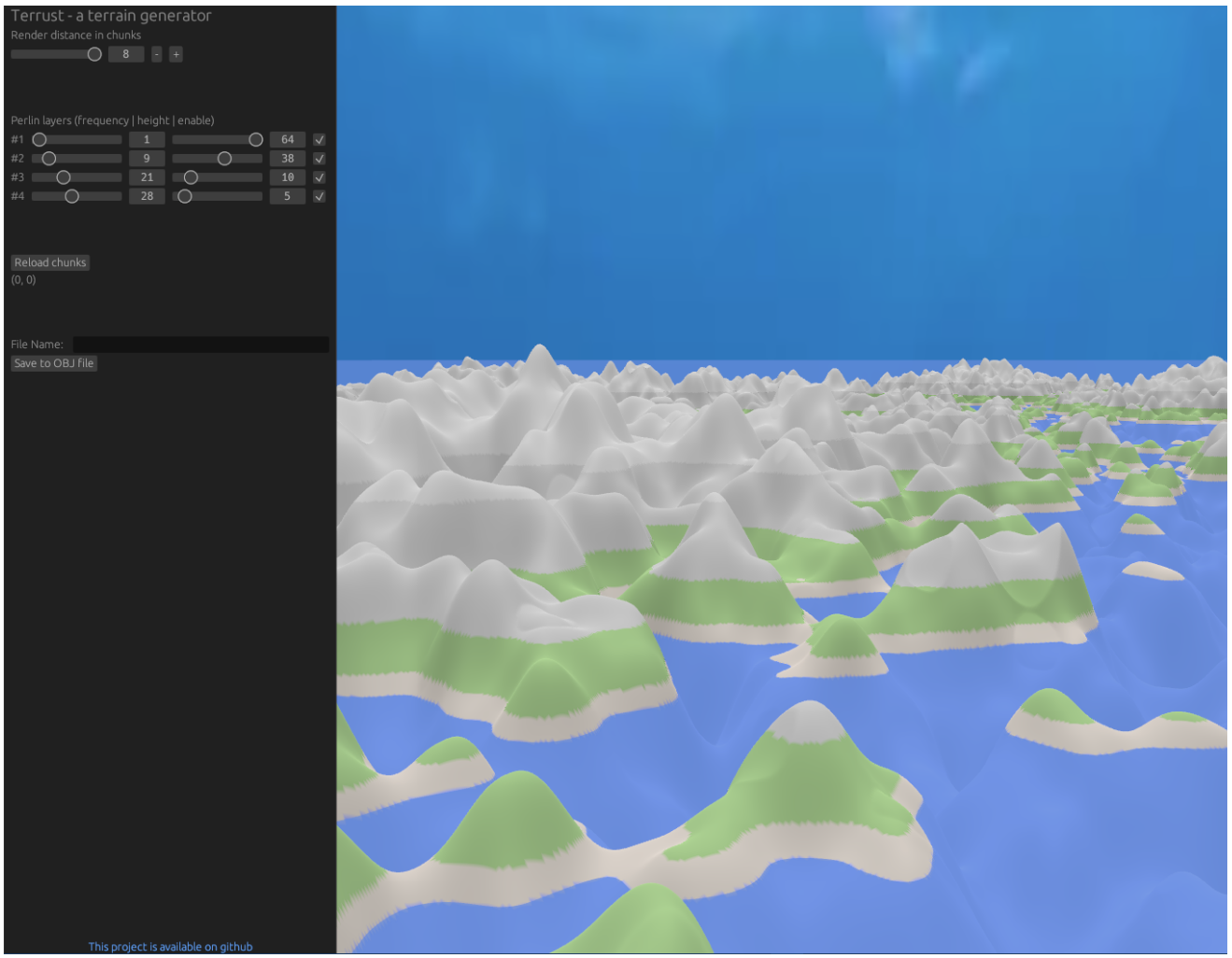
Large scale procedural terrain generation

The ■■■■■■■■
Centre Number: ■■■■

Terrust - a terrain generator in Rust



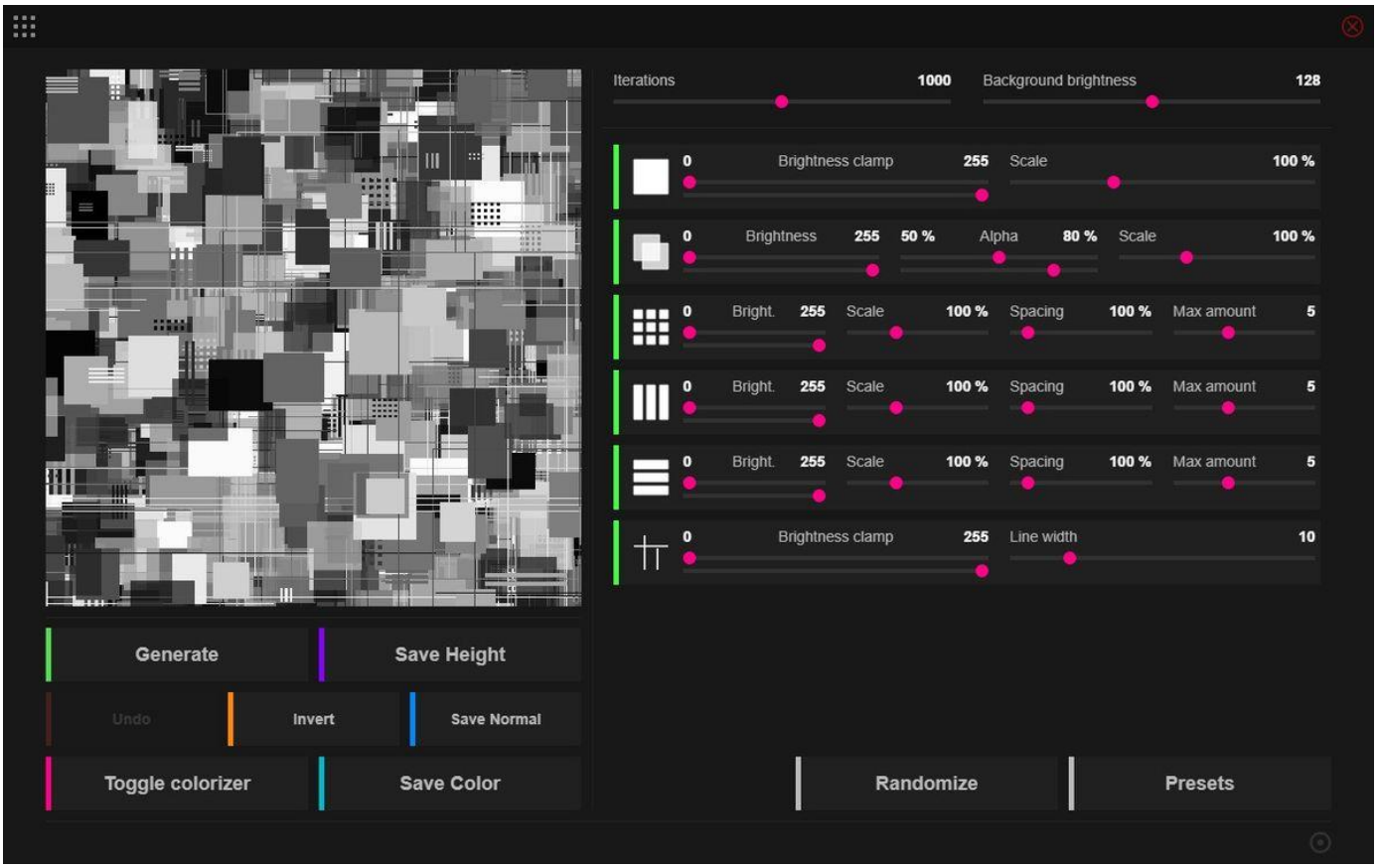
CONTENTS PAGE

Analysis

Introduction

The impact of computer generated graphics on the entertainment industry has been immeasurable. In the last 10 years the advances in computational technology have enabled creators to produce bigger and more realistic scenes, environments and 3D models for games. Even though computational powers allow for more complex environments the human resources still stays the same, as one human will only be able to produce what one human would. That is why in the recent years of 3D modelling the procedural generation started to get more recognition. Procedural generation is an algorithmic way of creating large amounts of data such as meshes or textures with input of controlling parameters from the artist, pre-maid rules and computer random noise. This method allows to produce massive amounts of data with very less than proportional amount of time spent on the production. Procedural generation is frequently used in texture generation where there is a need for a high resolution image with high density of details. In this project I am going to focus on the development of terrain generation software. This software would automatically create a life-like 3D model of the earth surface like pits, riverpaths and mountains.

There are a wide variety of software available for procedural generation, that includes standalone applications like PlacementJS or WorldGen as well as integrated plugins for 3D editors like OneClickDamage for blender and Unity.



JSplacemt – texture generator

Blender plugin – One Click Damage

But procedural generation is not limited to creative software, a lot of games utilise procedural generation for world generation. Most open world games use this technique as well as RPG dungeon explorers where there is high value for replay ability as every time the level is generated randomly and it presents a challenge for the player to face a completely new environment. One of those open world games is Minecraft, in this game the open sandbox world is generated algorithmically and could go on for nearly infinitely. The in-game generator can generate mountains, biomes such as deserts or rainforests and even things like villages and abandoned mansions automatically. In general procedural generation solves the problem of tedious manual modelling, what could be a nightmare to model by hand is now automated with the help of various algorithms.

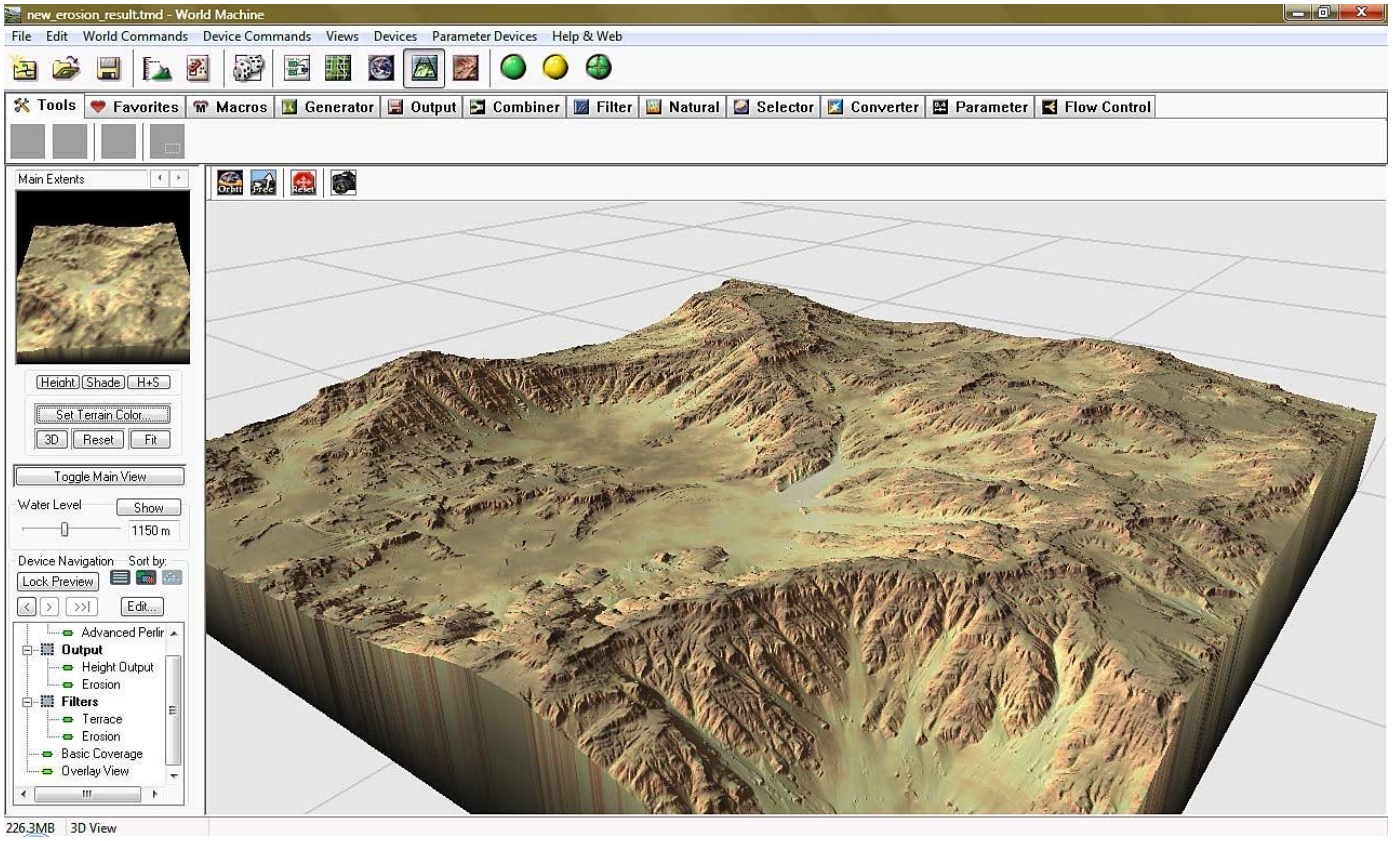
Minecraft [Game] – Map of random world

However because of being a relatively new tool in the industry, a low number of options exists in the market. Of course that is tied with the complexity of the problem, that is no easy feat to create a method which would consistently output data of acceptable quality. This is why most of the options that are available are heavily priced, and those that are not have limited functionality. This is natural as more money is needed for the high quality generator, but this puts a massive paywall between a struggling artist and avid amateurs before the exploration of procedural generation.

Overall, procedural generation software that is available for the general public today is either expensive or limited in capabilities.

Research

Presently there are quite a number of terrain generators in the market. They vary in functionality and price, but mostly the general workflow stays the same, tweak some parameters and preview the result. All of them feature a 3D viewport – a 3D window with movable camera which allows previewing the result of the generation. This would be a crucial feature to have in the project.



World Machine generator

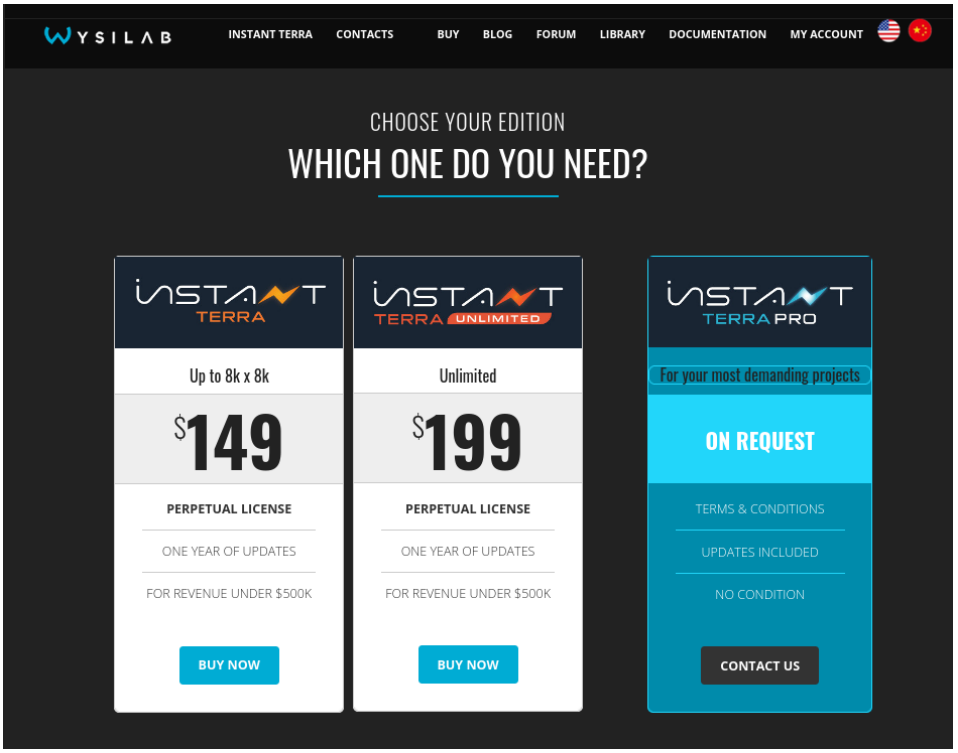
Most of the generators feature a large number of generation methods like soil erosion, various filters and textures. Unfortunately due to the time constraints of the project I would be unable to implement a large number of features like previously mentioned.

All of the explored generators are able to export the 3D model into the popular 3D file formats like OBJ, STL or glFT, by using such file formats it is possible to achieve a compatibility between various pieces of software across the industries as say potential environment artist could export their creating into OBJ file and handle it to the graphic designer, whose software would be able to recognize such file format. The process of exporting would consist of reorganising data into required format and writing it to the file, most of the popular programming languages feature libraries like that, they take in the vertex and indices data and save the model into the specified file format.

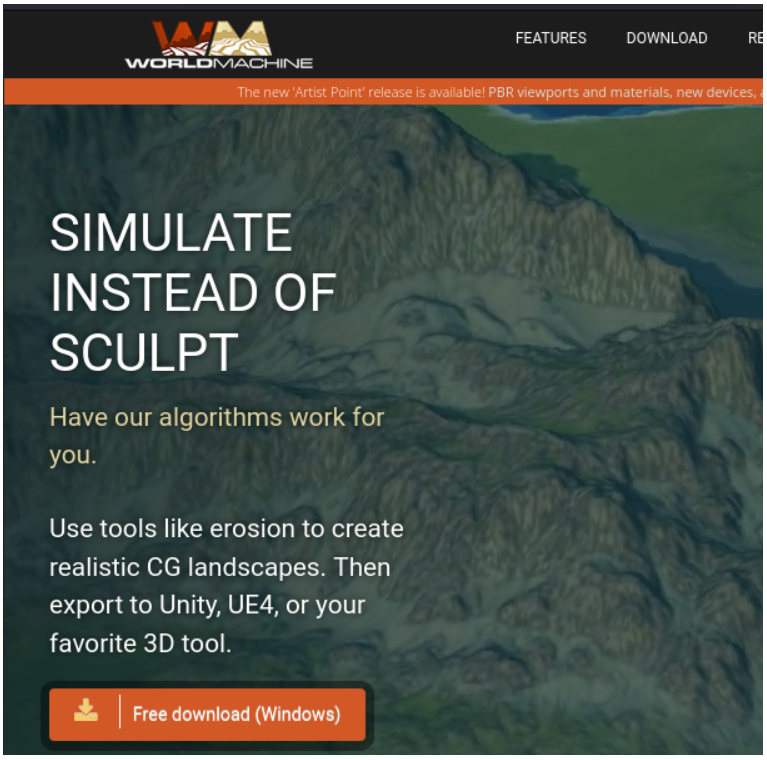
Across all of the generators one of the most popular methods of terrain generation would be Perlin noise layering. Perlin noise is an algorithm which allows to smooth out the continuous random data into nice random “clouds”, If multiple sizes of those clouds are stacked on top of each other then with a bit of parameters tinkering a mesh resembling a terrain will be formed.

However, one area that in my opinion could be improved is terrain generation. Currently there are plenty of terrain generators available, however the biggest turn away is that this field is heavily commercialised and most of the great utilities are marketed under freemium pricing strategy where additional extended functional and higher export resolution are behind a paywall. Being a student that I am that makes large scale terrain generation a hard area to dive into. Beside that the free available alternatives are often proprietary, they

don’t have their source code public, thus making it impossible to modify, this is a driving factor in software development as peer reviewed public software is most oftenly more secure and advanced as a collective effort of every programmer brings to the project. As well as that it would be great to provide application builds for other operating systems like Linux or MacOS, most common one of course is Windows, but it would be great to have a software that everyone could use, regardless of their OS preference.



(wysilab.com – Instant TERRA – Only perpetual licence available)



(World-Machine – Only available on Windows)

Interview

Questions for the potential user

1. Could you introduce yourself?
2. Have you used tools such as procedural generation for your works? If so, what did you use?
3. Were those programs free to use?
4. Have you had an idea of modifying it to suit your needs?
5. What do you think are the most important features that a software like that should have?
6. What did you not like about this software?
7. If you could edit the source code of the program what would you add?

Interview with a potential user

My interviewee ■■■ is in his 4th year of digital arts at the ■■■■■ in ■■■■■. I know that he works with 3D art on a daily basis, so I decided to hear his opinion on the current situation with generative software in the industry as well as ask for some suggestions he has for the project. He agreed to be part of the interview and I asked him the questions in the email. Here are his responses (Translated from Russian).

1. “Yes! Hello, my name is ■■■ and I am a bachelor studying digital arts at ■■■. I work with 3D editing software every day, as I am working part time in the Indie game studio as an environmental artist.”
2. “Yes. I mainly use Blender and ZBrush for my workflow, I have a lot of modelling plugins installed, like One Click Damage for Blender or Terrain Tools for ZBrush, the latter is what I think you are interested in as it can generate terrains procedurally, you just tweak the settings and press the button and it does all the job for you.”
3. “Well in general it depends on the software, for example in Blender most of the plugins are free, with some exceptions, but in other programs like ZBrush or CInema4D they are more often than not are paid. However, plugins that I need for my work are paid for by my employer, so It's not that bad.”
4. “Once yes, I had this issue with a Terrain Tools in ZBrush, I was modelling a background environment for the level and the plugin was crashing my project, I am pretty sure it was resolution of the mesh I was generating, so I had to use our company’s remote server with dedicated GPUs to generate the mesh.”
5. “I think that preview is one of the most important, I always tweak the parameters a lot during the modelling process and look around in the viewport to see the changes, so it really slows me down when I have to wait ages for the whole mesh to regenerate. I think the second most important is compatibility, I had this one case when my colleague sent me a draft of their work for feedback and it was a World Machine project file, I could not import it to Blender as it is not natively supported, so I had to ask them to export the mesh for me. When the work is done in teams I think that compatibility is the second most important feature.”
6. “The mentioned World Machine as an example has a resolution export limit for the free community version, so in order to use of for high resolution exports you have to pay, I get that it is made for the companies to pay, but then ordinary amateurs who can't pay large sums do not have access to this feature, so I think it's quite unfair.”
7. “Well to begin with there aren’t that many programs that have their source code available, but If I had a chance I would add something like a limiter for mesh generator, because my PC starts micro freezing sometimes as generation takes quite a bit of time.”

Feature discussion

From the interview with a potential stakeholder we identified the crucial requirements of the software functional. From what ■■■ was saying he is clearly not content with generators being so expensive,so as a project I will be doing it for free and releasing the source code to the public, so potential users would be able to modify things they would like and use it for free if they feel like they need additional functionality. This is quite

easy as there are a lot of free repository hosting websites like GitHub and GitLab which I could use to release the source code.

As well as that the app should have a 3D preview window of the model, so that any changes could be seen in real time and explored, additionally this preview frame rate should be stable, so that there is no stuttering and freezing during the movement and generation of the mesh. This would require the programming language to be a compiled one, as interpreted languages not only suffer in real time performance but lack in 3D engine support, a good choice for a performance focused language would be compiled languages like C family or other.

As ■■■ has mentioned the preview should be pretty big so that it is possible to preview multiple versions of the mesh until the good one is found. Well it would be impossible to generate the mesh of infinite size, but by using procedural generation it is doable to generate mesh on the fly, when the viewport camera will be nearing the end of the generated mesh a subroutine will be called to extend the mesh size and display a new version.

And finally one of the most important requirements is compatibility. By using a universally recognized file format I will make sure that it is compatible with other 3D editing software.

Requirements Gathering

By conversing with a potential user, I have identified software requirements.

1. Software must be free and open source.
2. The app must have a smooth 3D viewport, this implies the acceptable level of graphical optimisation that would enable a computer with average performance capabilities to display preview at a stable framerate with no major stuttering and freezing.
3. There must be a special setting to limit the size of the generated terrain.
4. The app must not freeze during the generation process and be able to preview the mesh freely with no major stuttering.
5. App must generate a high resolution mesh for export. In addition to that the app must support universally recognizable mesh file format.

Proposed solution

Implementation

This project is going to solve the described flaws as well as add some features to the generator. The application itself is going to be written in the performance focused programming language – Rust is going to be used as it is one of the fastest languages available and I know it already.

The software is going to fulfil its role as terrain generator by using a procedural generation of mesh. As the limitation of a single person project it would be impossible to achieve state of the art industry level of options available in terms of generation algorithms, soil water degradation simulation too is a project of its own and is not going to be included.

Compatibility

The application is going to be written for the target audience of 3D artists and game developers, so it is crucial to be compatible with popular 3D editing software. This is going to be achieved by using a universal

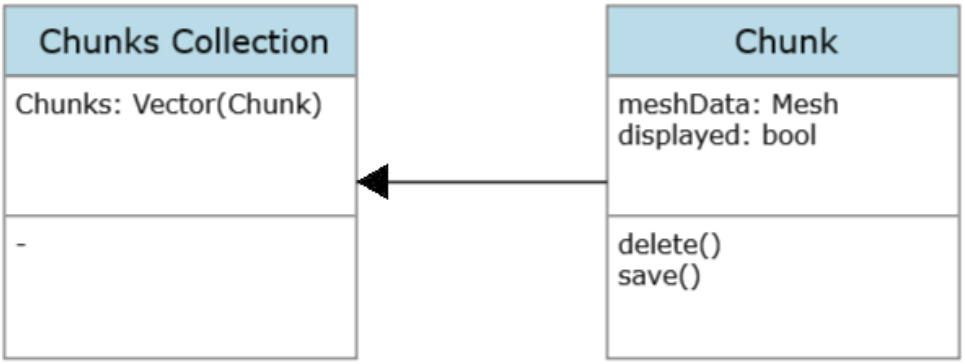
mesh file format that the app will export. According to the publication “Essential Guide to 3D File Formats” of Sonia Schechter[1] on her company’s website that specialises in 3D technology some of the most popular 3D file formats for 3D commerce are as follows:

- USDZ (Proprietary AR data format, created by Apple and Pixar)
- OBJ (Widely spread simple geometry data format)
- FBX (Popular for mesh and animations)
- gLTF (Popular in AR, VR)
- GLB (VR)

I have decided to use the OBJ file format as it is the most recognizable thus has the most compatibility with other software. As this format was created in 1990 and then publicly released in 1995 it is guaranteed to be compatible with 3D editing software, even old ones.

Abstraction

As the application would be communicating with a 3D engine as well as the 3D file export library, it would be beneficial to add a certain degree of abstraction like OOP. Mainly the terrain itself would be a simple array or vector of instances of chunks. Chunks themselves would have the mesh data as well as the visibility flag, as it would crash the program if we tried to display the mesh data that does not exist yet. As it is required to export mesh data, Chunk should have an export function that will save the data to the file as well as a delete procedure to safely remove it from the game world when the terrain is regenerated..



[1] https://www.marxentlabs.com/author/soniamarxentlabs-com/

User input

Upon starting the program the user will be presented with a 3D viewport and a sidebar which will have buttons to control the generation and export the mesh. Upon the input of the parameters they will be saved into a shared global variable so that all of the functions are able to access it. When parameters are inputted, the user will need to push the update button which will update the mesh of the terrain.. At all times the user will be able to push Esc to enter the movement mode, upon entering movement mode the mouse movement will control the virtual camera and WASD keys the camera in space. When the user finds the appropriate chunk, they would be able to switch back to editing mode by pressing Esc once again, regaining their mouse control, and save the mesh by inputting the name they would like to use and pressing the save button.

User will be using their mouse to drag sliders to adjust the parameters like render distance and noise settings. Once they are content with the result they will enter the text field that would be the name file and press save. The controls are going to be industry standard WASD movement with mouse. When the user moves their mouse the virtual camera will rotate about the X and Z axis. When W/A/S/D buttons are held on the keyboard the camera will start moving, if the user wants to move the camera faster they can hold the Shift key.

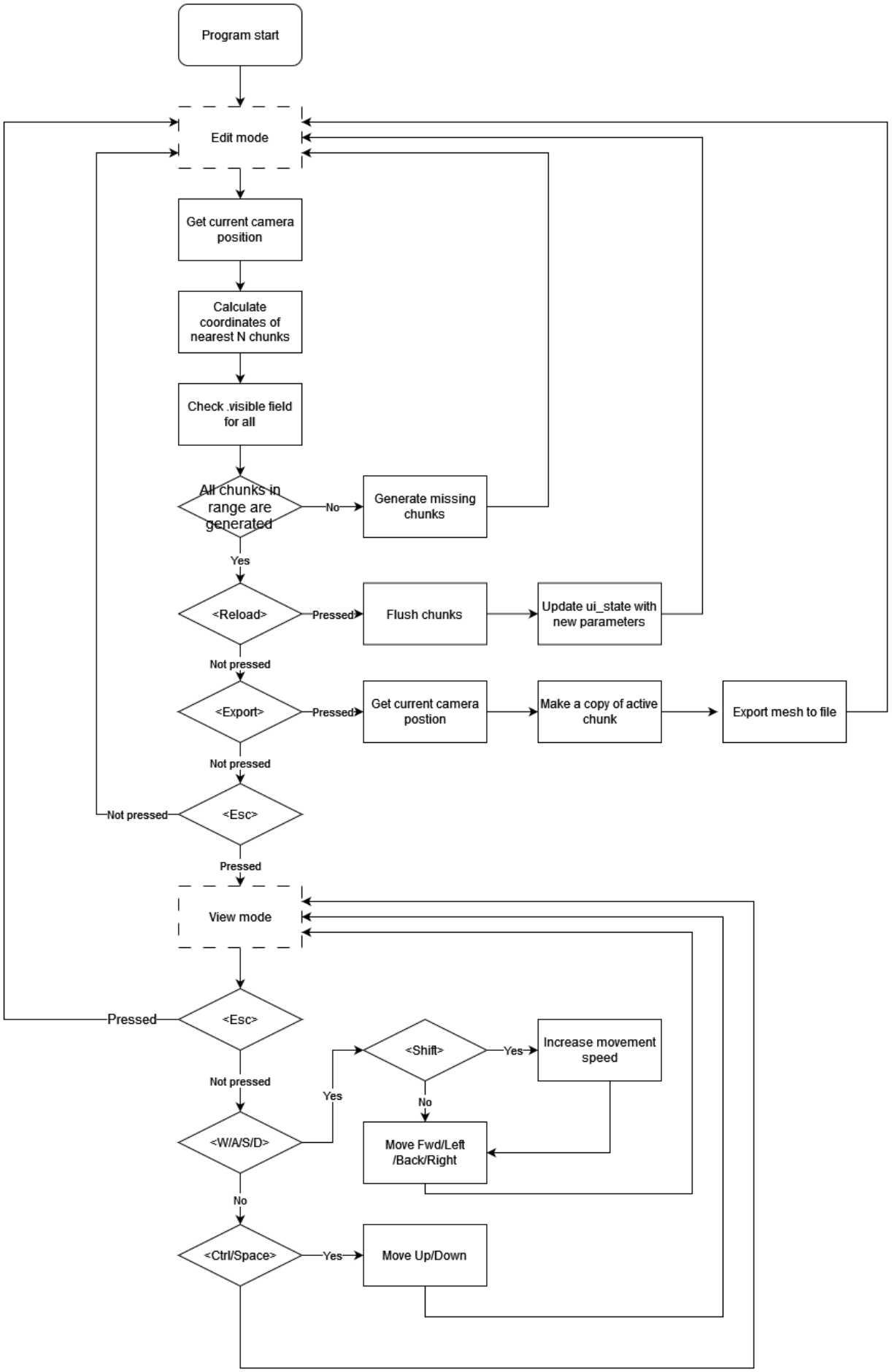
Key	Type	Description
Render distance	Unsigned Integer	Defines how far the mesh will be generated.
Noise inputs x4	Float	Perlin noise parameters such as frequency and height multiplier
Escape button	Toggle	Toggle movement and edit mode
W/A/S/D/Ctrl/Space buttons	Hold the key	Only in movement mode: move Forward/Left/Backwards/Right/Down/Up
Shift	Hold the key	Only in movement mode; pressed simultaneously with W/A/S/D to increase the movement speed
Mouse	Move mouse	Only in movement mode: rotate camera around 2 X and Z axis
Update	Press the button	Regenerates the mesh with new parameters
File name	String	File name to export the mesh to
Save	Press the button	Takes the string from File name field and saves the chunk to the OBJ file

Program logic

The overall logic of the program could be summarised as such.

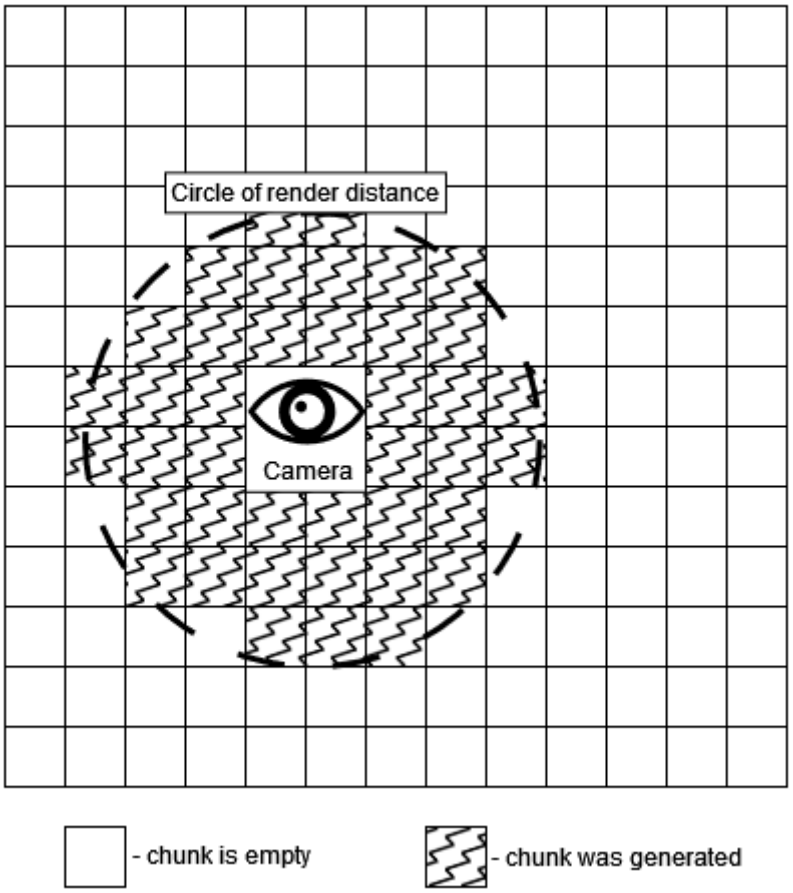
- User starts up the application
- They tweak the parameters
- Press the update button to regenerate the terrain
- Press the Escape button and navigate around the world to preview the mesh
- Leave the movement mode by pressing the Esc button
- If necessary tweak the the sliders to achieve the desirable terrain shape, update the mesh
- Input the name of the file ending with “.obj”
- Press Save button

Under the hood however a bit more should be done to ensure that all of the objectives are met. Once the user enters the movement mode, the app will be checking the current location of the camera to see if the terrain should be expanded further according to the render distance the user has set. And finally, to make sure that the app is not taking too much resources that may result in freezes and stuttering periodically the collection of chunks will be deleted from the world. If a user moves too far away from the generated chunk and does not see it, it would be better to have it gone from the memory to free up memory resources, so every minute a procedure will be run that will delete the collection of chunks. Because visible chunks are checked constantly the deletion will be instant as the generation of deleted chunks near the camera, so the user will not be able to notice it.


Program start
Edit mode
Get current camera position
Calculate coordinates of nearest N chunks
Check .visible field for all
All chunks in range are generated
No
Generate missing chunks
Yes
<Reload>
Pressed
Flush chunks
Update ui_state with new parameters
Not pressed
<Export>
Pressed
Get current camera postion
Make a copy of active chunk
Export mesh to file
Not pressed
<Esc>
Not pressed
Pressed
View mode
Pressed
<Esc>
<Shift>
Yes
Increase movement speed
Not pressed
Yes
No
<W/A/S/D>
Move Fwd/Left /Back/Right
No
<Ctrl/Space>
Yes
Move Up/Down

Procedural generation

When the update button is pressed, it will update the noise parameters and flush all of the chunks in the collection. As previously mentioned the checking for visible chunks is being performed constantly so the chunks around the camera will be generated if they are in proximity which is set by the render distance.

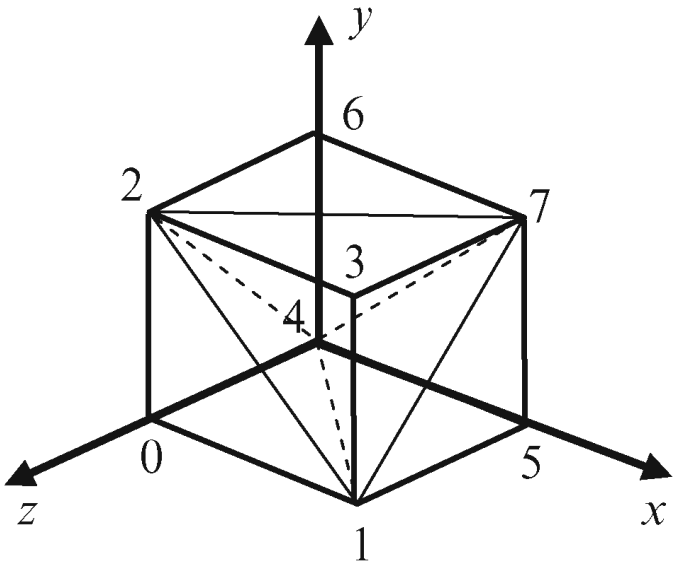


Mesh generation

Vertices and Indices

Basically every chunk will have its mesh data procedurally generated. For example to describe a mesh of a simple cube in a 3D space a Vertex-Triangle notation will be used. Each point will have 3 values x,y and z offset from the world origin. By themselves vertices are nothing but disconnected points in space. Triangle list is used to describe the connection between them, each index is pointing at exactly one vertex in the list.. In computer graphics the most primitive shape is a triangle, to define it 3 points are needed as well as 3 index numbers. For example 2 bottom facing triangles of a cube will be:

- Triangle 1
  - Vertices
    - (0, 0, 0)
    - (0, 0, 1)
    - (1, 0, 1)
  - Indices
    - 4, 0, 1
- Triangle 2
  - Vertices
    - (0, 0, 0)
    - (1, 0, 1)
    - (1, 0, 0)
  - Indices
    - 4, 1, 5



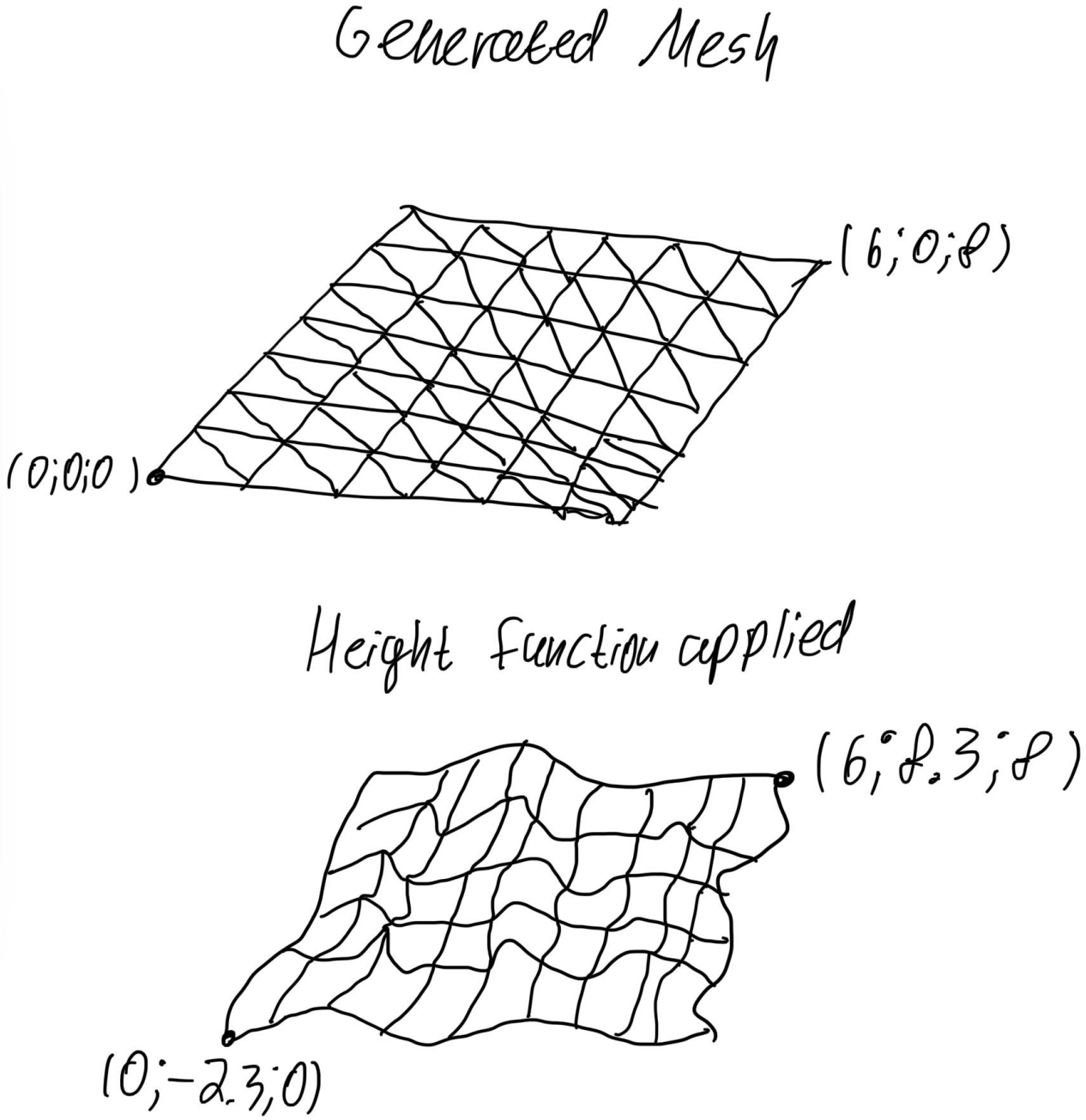
Vertex List		
x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

Triangle List		
i	j	k
0	1	2
1	3	2
2	3	7
2	7	6
1	7	3
1	5	7
6	7	4
7	5	4
0	4	1
1	4	5
2	6	4
0	2	4

The position of the (0, 0, 0) in the Vertex List is 4, so index 4 is pointing at it, (0, 0, 1) position is 0 and so on. Notice that the indices go counterclockwise when defining a triangle, this will be important later during the shading of the mesh.

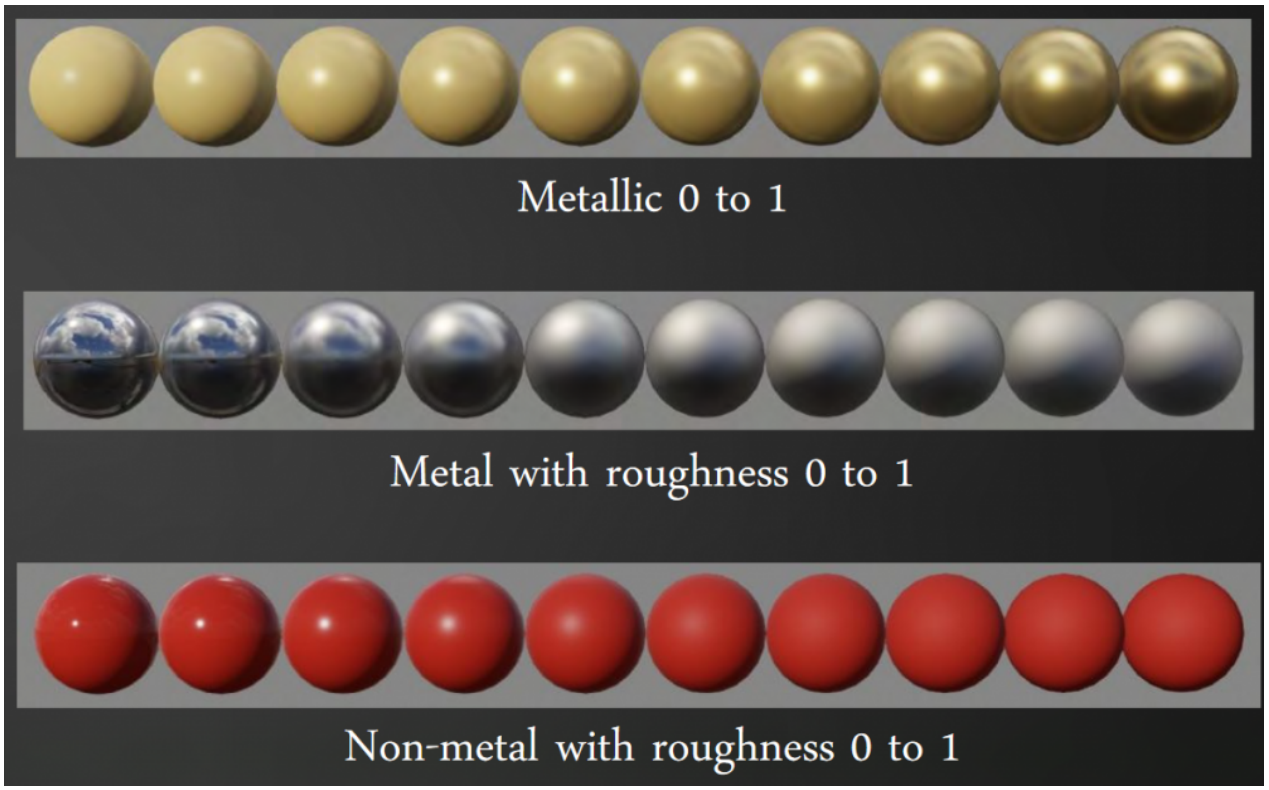
Offset function

In case of the terrain the mesh would not be a cube but a grid – a collection of evenly spaced points across the X and Z axis, the Y value of each vertex will be zero for now. The height function will generate the Y values for each vertex and when all of the Y offsets are generated the flat plane will become displaced.

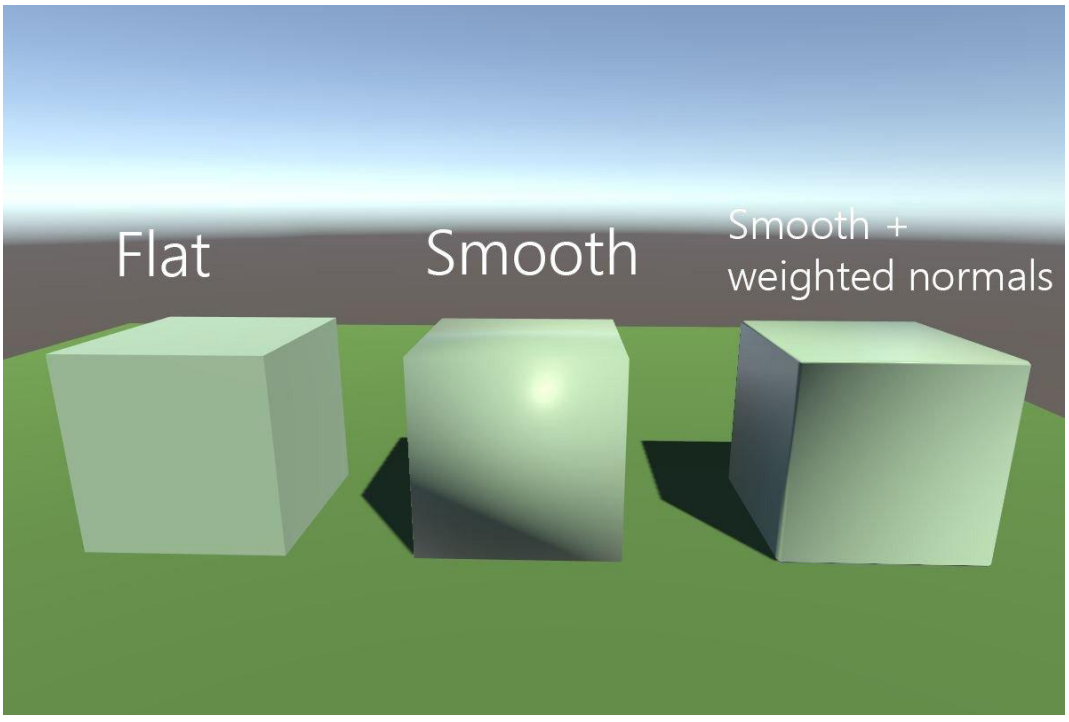


Lighting and Normals

Next up is correct lighting. Each mesh has normals, normals are direction vectors that are used for PBR – physically based rendering, it is a clever computer graphics technique that attempts to imitate how light and matter is seen in the real world. The metallic objects will have light reflections, rough objects will have light scattered and so on.

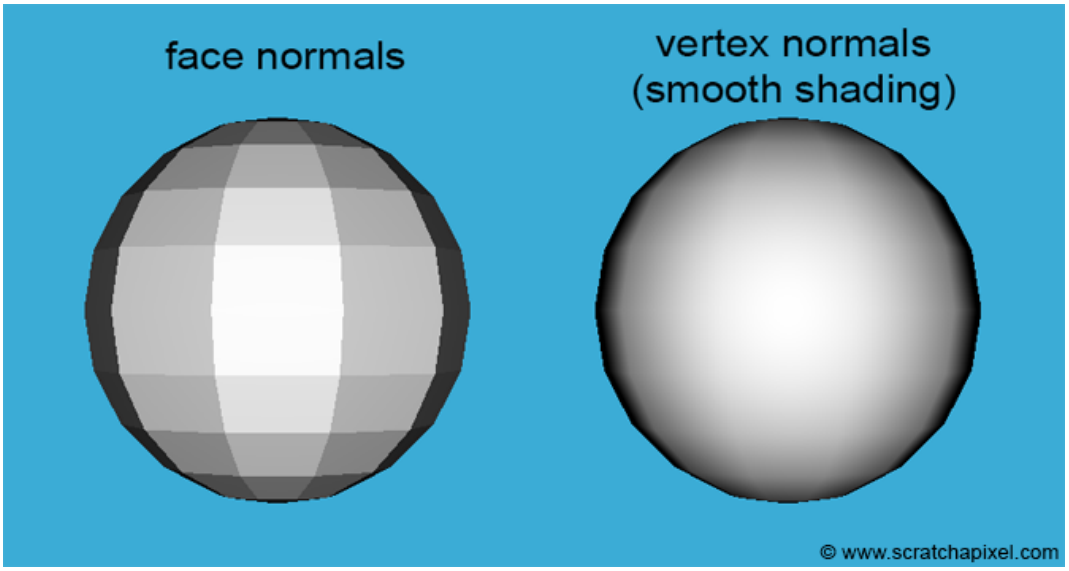


In order to correctly reflect, scatter and cast light mesh should have its normals pointed in the right direction. There are several ways to calculate the normals for each vertex.



- Flat (face normals) – each normals is perpendicular to the nearest triangle in the vertex list.
- Smooth(vertex normals) – The normal is calculated by taking the average of normals to 3 adjacent triangles that the vertex takes part in
- Smooth weighted – same as Smooth, however when the averaging is performed individual areas of triangles are taken into account.

Generally flat normal methods will look more choppy and less pleasant, so it should be avoided, as preview of the terrain should be smooth and pleasing. The recalculation of normals will occur after indices and vertices are generated.



For the described method normal calculation quite a bit of vector maths is going to be used such as finding normal to plane, calculating cross product and finding the weighted average.

Miscellaneous

For comfortable preview the mesh is going to be coloured using a procedurally generated shader. This shader will take the height value of each vertex and colour the area around it according to the colour ramp that would represent water, sand, grass, rocks and snow.

As well as that, for the comfortable preview it would be beneficial to have a water level present, it would be a simple plane of a blue colour.

Final Objectives

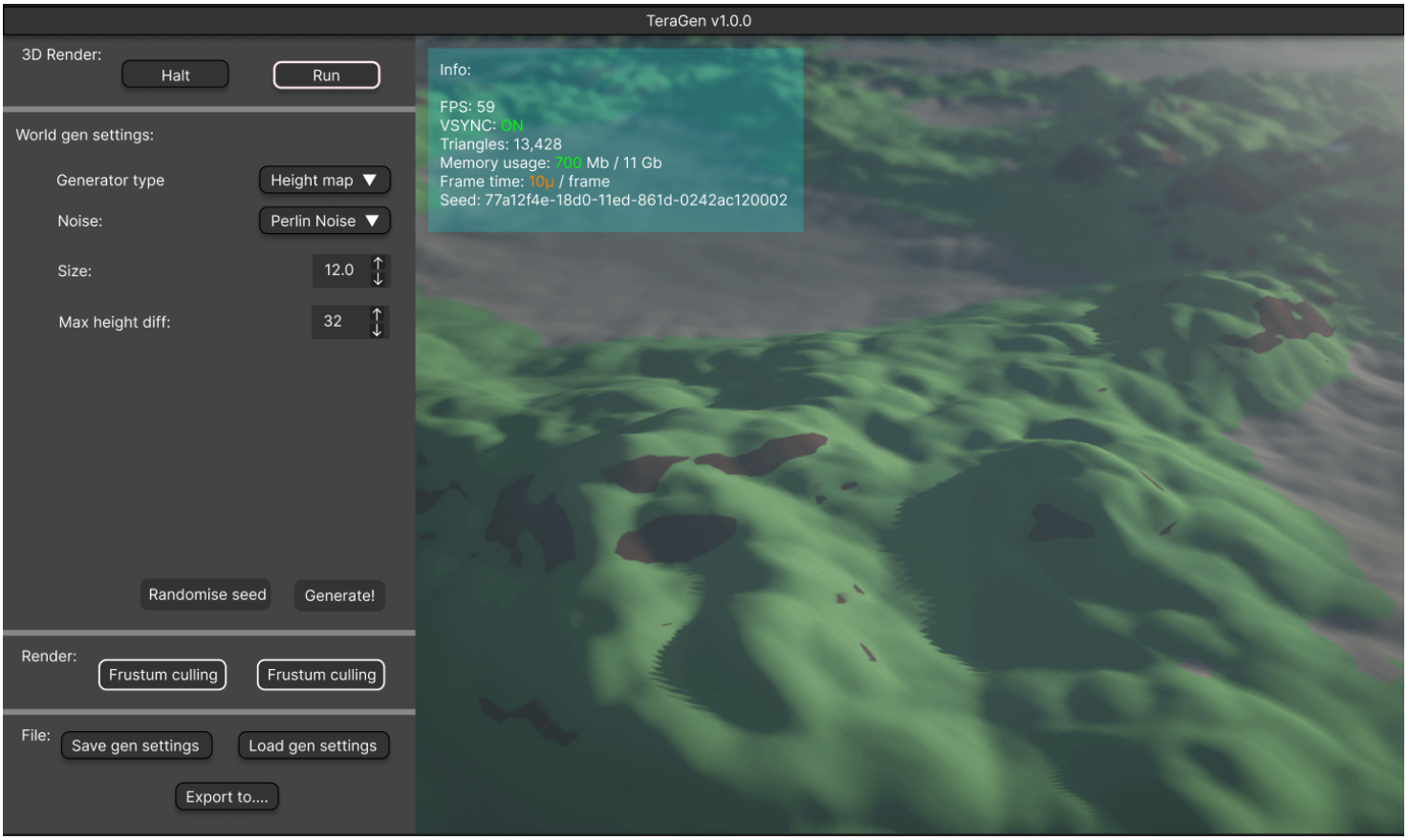
1. Software type (Application must be)
    a. Free
    b. Open source and available online
    c. Build available for Windows and Linux
2. Interface
    a. Window with a 3D preview of the generated mesh
    b. The viewport camera can be controllable by the user
    c. Preview is smooth and does not stutter when camera is moved
    d. There is a big area for mesh preview
    e. Render distance must be adjustable
    f. After parameters are adjusted user should be able to regenerate the mesh
    g. Shade the mesh smooth for comfortable preview of the mesh
    h. Mesh is coloured according to vertex height
    i. Sky as a background
3. Mesh
    a. Mesh is algorithmically generated
    b. The generation parameters are adjustable
    c. App must export high resolution mesh
    d. Save button exports mesh to OBJ file

Documented Design

Overview of design

UI

The application is going to be in a window with a 3D view port and options to the side. There will be multiple tweakable parameters for mesh generation as well as multiple buttons. The viewport will have a flying camera controlled by the user. The mesh is going to be procedurally coloured according to the vertex height, so that lower parts are blue (water) and upper parts are white like snowy mountain tops.



(Design sketch in Figma)

Language

There are several requirements for the language and render engine. Firstly we should be able to draw a window for UI, that is not a novelty as most if not all programming languages have frameworks for that. Generally the terrain generation is going to require a lot of computation being performed moreover in real time. That is why compiled languages will be more fitting for this task. The language should have a GPU API such as OpenGL or Vulkan to enable 3D rendering of the mesh and if possible calculation offloading. C++, C# and Rust sounds like a good fit! I am going to choose Rust as it has in-built memory safety which will help prevent bugs later down the development. As well as that Rust has a variety of 3D engines to choose from, Bevy game engine seems to be the best option.

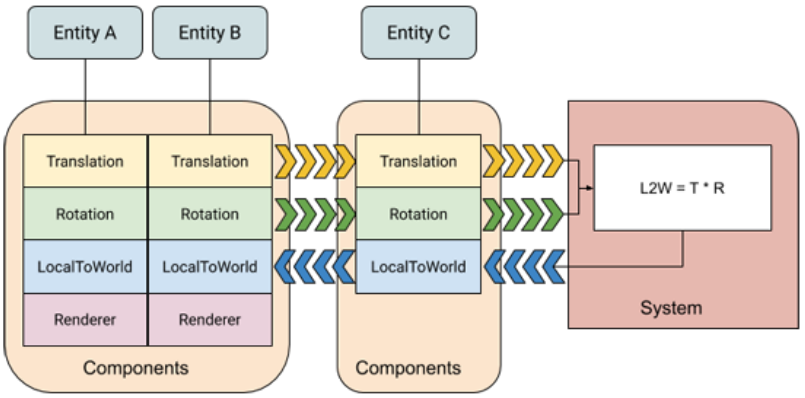
ECS

Bevy follows the Entity Component System[2] (ECS) which is the software architecture pattern that is used in most of the game engines. This system allows for easy registration of Entities in the code which are basic structures or types like Player or World. Components which are the particular fields of the Entities like player name or terrain texture and finally Systems, those are the processes that manipulate Entities' Components following certain rules like colour of the sky changing if it is night time or objects having mass and falling if they are thrown. In our case the:

main **Entities** are going to be ChunkResource and UiState, those will be modified in real time

Components will be very few, as all of the present data is already defined in the Entity class, only one component will be needed – MeshIndicator, it will be used to identify mesh blocks (chunks) by giving them a MeshIndicator component, which would then be easily queryable across the whole system

and finally **Systems** such as chunk update and flush. Chunk update will update all chunks around the active camera according to the parameters defined in the UIState which will supply noise parameters as well as active render distance in which those chunks should be generated. Chunk flush system will be periodically called to despawn old mesh data, as there will be too many mesh to display the program will start to stutter, so periodic flushes will ensure stable framerate during runtime

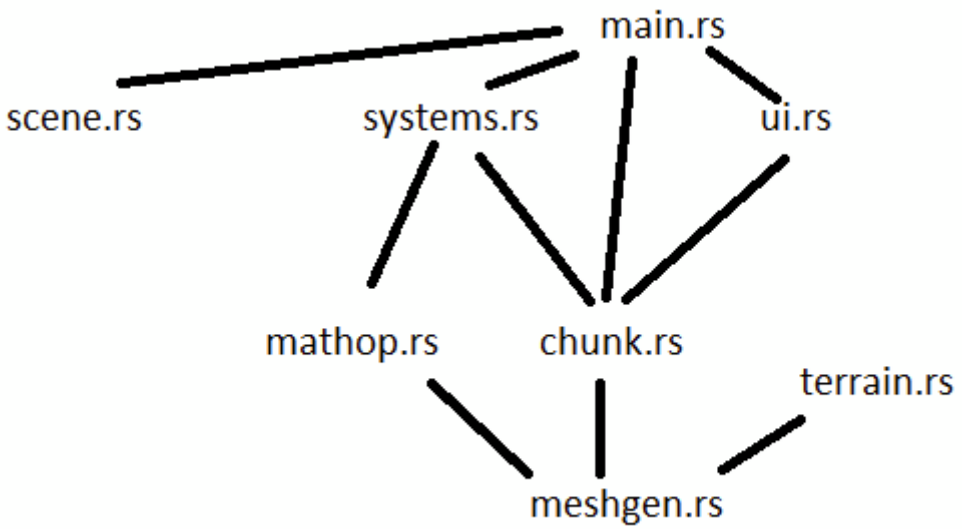


Sample diagram of generic ECS world

General file structure

As this project involves a lot of different abstraction and classes to be implemented it would be beneficial to split the whole structure into small files, separately defining their own purpose. For example the main.rs will only have initial app start and window settings, chunks.rs will define the ChunkResource and Chunk structure and mathop.rs all of the commonly used mathematical operations.

[2] Entity component system - Wikipedia



main.rs	main project file that includes the library imports, general settings like window scale and the startup structure of bevy
scene.rs	setup procedures of the main scene such as water plane, skybox and the viewport camera
systems.rs	ECS systems such as chunks flush
ui.rs	defines entirety of the user interface and enable the dynamic tweakable parameters
mathop.rs	a collection of mathematical and vector operation used throughout in the project such as vector normalisation cross product, etc
chunk.rs	describes the class and methods of Chunk structure such as chink visibility, chunk export and chunk dynamic update system.
meshgen.rs	includes all of the functions that generate the mesh: skybox, chunk and its generator functions like normals and offset
terrain.rs	defines the generation algorithm that is going to be used for the offset generation

Main.rs

Will have imports such as bevy library as well as external camera plugin. Then the constants will be defined such as default render distance and chunk scale. After that in the main function the bevy app will be initialised with App::new(). It should include the settings for app window like height, width and name, user interface layout defined in ui.rs, shared resources like UiStates and CunkResource, chunk update system, viewport camera plugin and finally the initialization functions such as scene setup and chunks initialization.

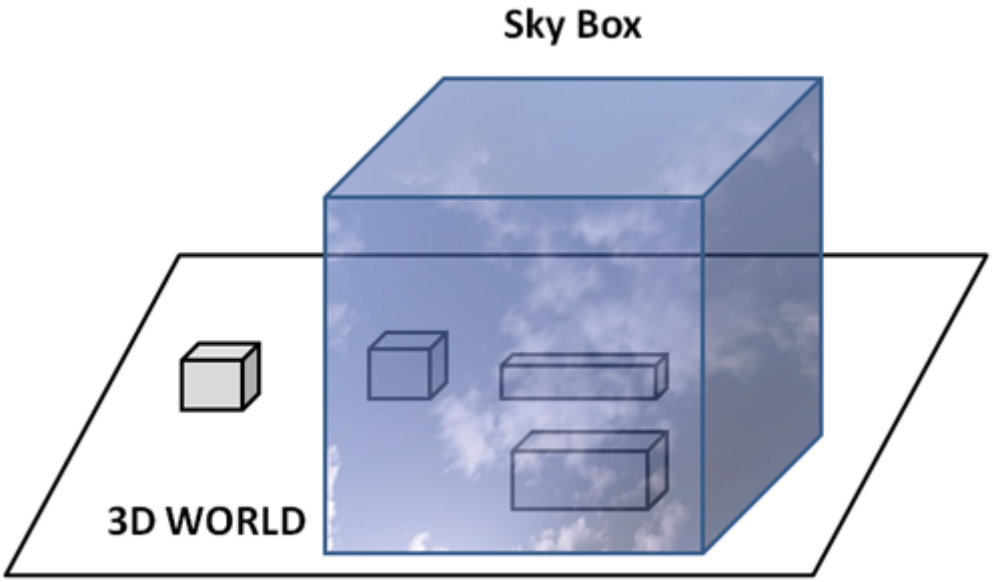
Function name	Parameters	Description
main	N/A	Imports library dependencies Initialises the Bevy App Insert Resources and Systems Start the application

Scene.rs

Will have a basic viewport spawn command which is a user controllable camera with a preview of the world as well as the main scene initialisation routine, it will take in the mesh resource, to insert new meshes to the project and asset_server, to enable textures for the skybox.

In order for the world not to be in complete darkness a global light will be added. It is a simple directional light which emulates the way the sun emits sun rays. It will have a high luminance value like 30000 and a yellowish shade.

As it is required to have a comfortable preview I decided to add a skybox to the world. Originally skybox is a cube that has a texture of a sky and is set apart from the scene, so that it creates an illusion of the sky being present.



Nowadays skyboxes are made with spheres. It is going to be generated by using the meshgen.rs function gen_skybox() which returns a UV Sphere. The whole scene is going to be inside it, so it would be big and have its mesh indices inverted with mathop.rs inv_idx() so that when the sky texture is applied it is going to be visible from inside. (in the modern gpu graphics by default each mesh has triangle direction, normally if it is seen from front then the indexes of points go counter clockwise, from the back side those triangles are not going to be visible unless the order of mesh indices is inverted to be clockwise, this way the UV sphere will have its indices triplets for each triangle reversed, for example a single triangle with indices [1, 3, 5] will become [5, 3, 1], so that the texture of the sky is visible from inside the sphere).

From outside of it the skybox sphere will appear as a sphere with a weirdly stretched texture.

Then the simple material will be initialised with the sky texture, it will be excluded from the lighting process as skyboxes are already lit by their texture.Finally the skybox bundle will be spawned with mesh as a UV Sphere and material of the sky.

After that the water is going to be created. Visually it is no different from a blue flat plane, so a plane with blue material applied to it will be added to the scene, because plane should not be visible outside of the terrain area the blending option will be set to Alpha Blend, meaning that the blue plane is going to be seen only if there is another material under it, another material being the terrain material.

Function name	Parameters	Description
basic_scene	1. Bevy Control Commands 2. Bevy Mesh Asset (for meshes) 3. Bevy Asset Server (for textures) 4. Bevy Standar Material Asset	Setups global light, skybox and water level

Systems.rs

This file will contain a system that will be running concurrently while the program is open.

As terrain needs to be reloaded every once in a while to discard the chunks that are too far away from the camera chunk_timed_flush is going to be defined, every 60 or so seconds it will access the ChunksResource and call the flush method.

While the user moves the camera chunks are going to appear in the render distance as discussed previously. In order to do that a second system is going to be introduced – chunk_update_sys this one will be called every frame. It will generate the coordinates of chunks that are needed to be displayed according to the render distance and check each individual Chunk in range if it is generated and drawn properly, if chunk was not generated, it will be added to the scene and the visibility field will be set to True.

Function name	Parameters	Output data type	Description
chunk_timed_flush	1. Time since program started 2. Bevy Query with MeshIndicator (see ECS on pg. 22) 3. Bevy Control Command 4. Chunk Resource	N/A	Flushes chunks every 60 seconds since the start of the program
chunk_update_sys	1. Bevy Mesh Assets 2. ChunksResource 3. Bevy Control Commands 4. Bevy Standard Material Asset 5. UIState 6. Bevy Query with Camera Indicator	N/A	First locates the integer coordinates of the nearest chunk. Then gets the coordinates of chunks in the render distance and goes through all of them one by one, if chunk is not yet generated, then generates it, sets .visible field to true and saves the chunk mesh to .mesh field
chunk_load_near	1. Tuple of integer coordinates – (X, Y) 2. Effective radius – R	Vector of i32 tuples	Returns the coordinates which Manhattan distance from (X, Y) does not exceed R

Ui.rs

Will have all of the visual user components defined. The bevy_egui library is going to be used as it is integrated in the bevy engine. Firstly a shared resource UiState will be described, it is a simple structure that will have all of the tweakable parameters inside it, so that when the changes are applied to say render distance, the chunk update system is able to read it safely. Then the default values for the UiState will be described in the ui_state_defaults, so that on application startup user can already see a working place with mesh and preview.

ui_system will describe the layout of the ui. It will be a side panel with a list of buttons and sliders that are going to update the global variables state. For example, if + or - buttons are clicked in the render distance section then an anonymous function will be called incrementing the ui_state.render_distance by 1 or -1, same goes for noise levels sliders.

The reload button will first update the new noise parameters then call the flush function which will despawn all of the entities on the screen, because the chunk_update system will always be running in the background the grid will be regenerated automatically in the proximity to the camera. First a query will be made to access the camera Transform component, it contains float32 coordinates of the camera in the world, then the integer conversion will happen to round the camera position to the nearest integer position in the world to generate chunk in, in order to avoid going out of bounds the negative values will be clamped to 0 with if statements, after converting the position of the camera in the integer form a function will be called – chunk_load_near, this function takes in a tuple of integer coordinates and a integer distance and returns an array of coordinate tuples that are within the specified distance from the tuple given, this way it is possible to obtain a near N chunks at any (X, Y) position, after this function is called the obtained tuples will be converted into 1-dimensional index that will point to the vector of chunks in the shared ChunkResource. As ChunksResource has a field .chunks which is a simple fixed size vector of type chunk it is possible to calculate the unique index of the chunk by taking the X coordinate and adding it to the multiple of Y and grid size, this way an index number will be obtained. During the initialization of the chunks field all of the chunks are defaulted to visibility set to false, which signals that the chunks are not displayed, this helps to determine which chunks are already present in the world and which are not and should be generated. So the result of chunks_load_near is cycled through and checked if the chunk at specified index is visible or not, if it is visible then for loop continues, if it is not then it should be generated.

In order to create a chunk in the world first its mesh is generated by calling gen_ter_mesh. THis function takes in a CHUNK_RES a global constant which described the resolution of the chunk, by design it is set to 1024, a coordinates of this chunk, in order to generate appropriate noise values and a copy of a ui_state structure which contains parameters needed for the noise generating algorithm. After the mesh is obtained it is bundled into the PbrBundle, a handy method which incorporates mesh, position and shader of the chunk. The transform field is a relative position of the chunk to the world origin, so previously generated integer coordinates are taken and multiplied by the global value SCALE, which defines the scale of the chunks mesh. The material field for now is defaulted to simple white colour PBR. After that the bundle is marked with a MeshIndicator component, so that it could be queried later by using this marker, for example for flushing, it would be possible to despawn all chunk mesh leaving skybox and water level intact. And finally the chunk field .display is updated to be true, so that it won't be regenerated in the next cycle and the mesh data is saved to the .mesh field for export functionality.

Function name	Parameters	Description
ui_state_defaults	UiState Resource	Initialises UiState with default recommended parameters.
ui_system	1. BevyEGUI Context 2. UiState Resource 3. Bevy Query with MeshIndicator 4. Bevy Control Command 5. Chunks Resource	Initialises Bevy eGUI side panel. Adds all of the control sliders and text descriptions to them. Adds buttons for update and export at the bottom.

Mathop.rs

Mathop.rs contains all the meth related functions that are used throughout the project, primarily consisting of vector geometry. For example to update normals for the newly generated mesh it would require to

calculate new direction normals according to the connected triangles, that task could be accomplished if normals are calculated for those 3 triangles by using a cross product formula, then averaged and normalised in order to create new normal direction.

Function name	Parameters	Output data type	Description
vec_sub	2 f32 vectors - A and B	1 f32 vector	Returns subtraction of A from B
vec_add	2 f32 vectors - A and B	1 f32 vector	Returns addition of A to B
cross_prod	2 f32 vectors - A and B	1 f32 vector	Returns cross product of A and B
tr_wt_avg	5 f32 vectors - O, A, B, C, D	1 f32 vector	Returns a weighted average normal (smooth normal) from triangles forming about O with A/B/C/D
tr_area	2 f32 vectors - A and B	1 f32 number	Calculates the area of a triangle that is formed with A, B and a Zero coordinate
vec_mag	1 f32 vector - A	1 f32 number	Calculates the magnitude of vector A
vec_scale	1 f32 vector, 1 f32 number - A, S	1 f32 vector	Multiplies vector A by S
inv_vec	1 f32 vector - A	1 f32 vector	Inverses the direction of the vector A
vec_norm	1 f32 vector - A	1 f32 vector	Normalises vector A

Chunk.rs

It contains all of the chunk related code. First the Chunk structure, it has 3 fields .mesh – mesh data for export and.visible – for efficient terrain generation around the camera.

The mentioned MeshIndicator is a blank structure that has no fields because it is only used as a marker for querying.

ChunkResource is a simple Bevy resource that would be accessible throughout the entirety of the project, it has only one field,chunks which is a simple vector of Chunk type.

It has multiple subroutines defined such as flush method, it queries the list of all object in the world for those that have the MeshIndicator marker and despawns them as well as goes through the .chunk field of ChunkResource to update the visibility of all elements to false, signifying that they should be regenerated.

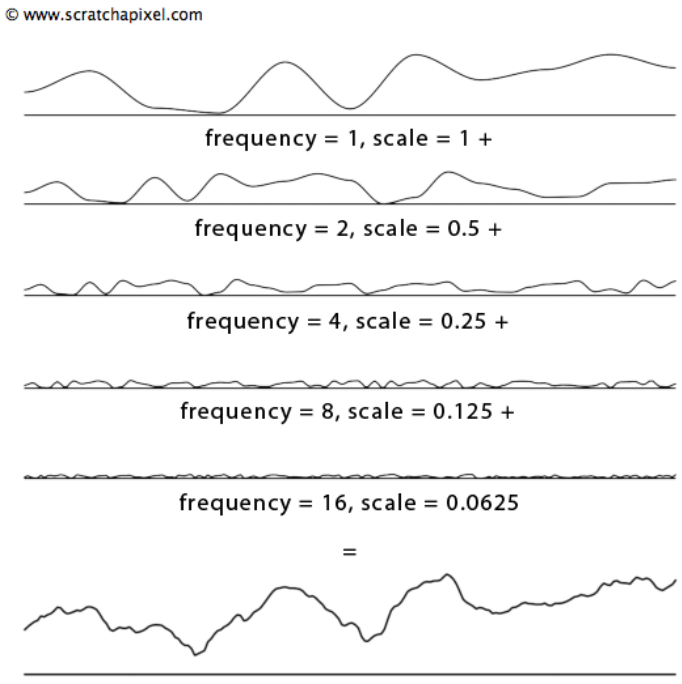
Next method is to save a subroutine, it takes in camera integer position and file name. By converting XY coordinates to vector index, the previously required Mesh instance is found, then the .mesh field is copied as it is needed to be modified before saving. Originally the array of indices is stored as a single big vector, but it is needed to be converted into the vector of arrays of length 3, so by utilising an additional vec_chunk function which simply splits a simple vector into vector of triplet arrays indices are reformed. And finally using the external library meshx the mesh is saved into the specified .OBJ file.

Function name	Parameters	Output data type	Description
ChunkResource.flush	1. Bevy Query with MeshIndicator 2. Bevy Control Command	N/A	Despawns all objects in the world that have the MeshIndicator marking and sets the .display field of all chunks to false.
ChunkResource.save	1. Tuple of integers – chunk location - (X, Y) 2. File name	N/A	Saves the chunk from the position (x, Y) to a OBJ file with name specified
vec_chunk	1 u32 vector - A	Vector of u32 arrays of size 3	Chunks the vector A into triplets. Produces a vector that consists of arrays of usize (u32) integers with fixed length 3
chunks_init	ChunksResource	N/A	Initialises the .chunks field with blank exemplars

Terrain.rs

The terrain.rs file contains the mesh generation algorithm which at the moment is a simple procedural Perlin noise. The gen_perlin function takes in the tuple of size of noise, offset as well as layer parameters for the generation, and returns a simple vector of 3D points.

Firstly a blank vertex array is generated with the size provided, it will be updated later. The standard Perlin library provides a perlin noise generator which takes in a tuple of floats which indicate a position of a vertex.



In order to generate new offset for vertices a multiple for cycles are used. Inside it x across goes along z which goes over every single vertex consequentially. For every vertex a unique offset is generated which is added to the variable, there is a total of 4 layers, so a new for cycle goes through the array of layer parameters which were provided in the arguments, the 4 perlin layers are effectively the same but their frequency and scale are different, so when the 4 layers are added on top of each other it creates a completely new terrain like mesh. So Perlin.get() method is called to get a unique offset for the specified coordinates which are scaled according to the layer scaling factor which changes the frequency of the noise as well as the scaled in the end, so that say bigger scale noise has more influence on terrain and it appears more natural. The 4 generated are added and recorded as a y offset to the vertex, this process is

repeated until every single vertex has an updated y coordinate. The function returns a vector of 3D points with a new generated offset.

Function name	Parameters	Output data type	Description
gen_perlin	1. Integer grid size -- (N, M) 2. f32 offset for all points - (F, G) 3. Vector of arrays of f32 triplets – Layers	Vector of f32 3D points	Generates points of layered Perlin Noise according to the parameters given in layers vector

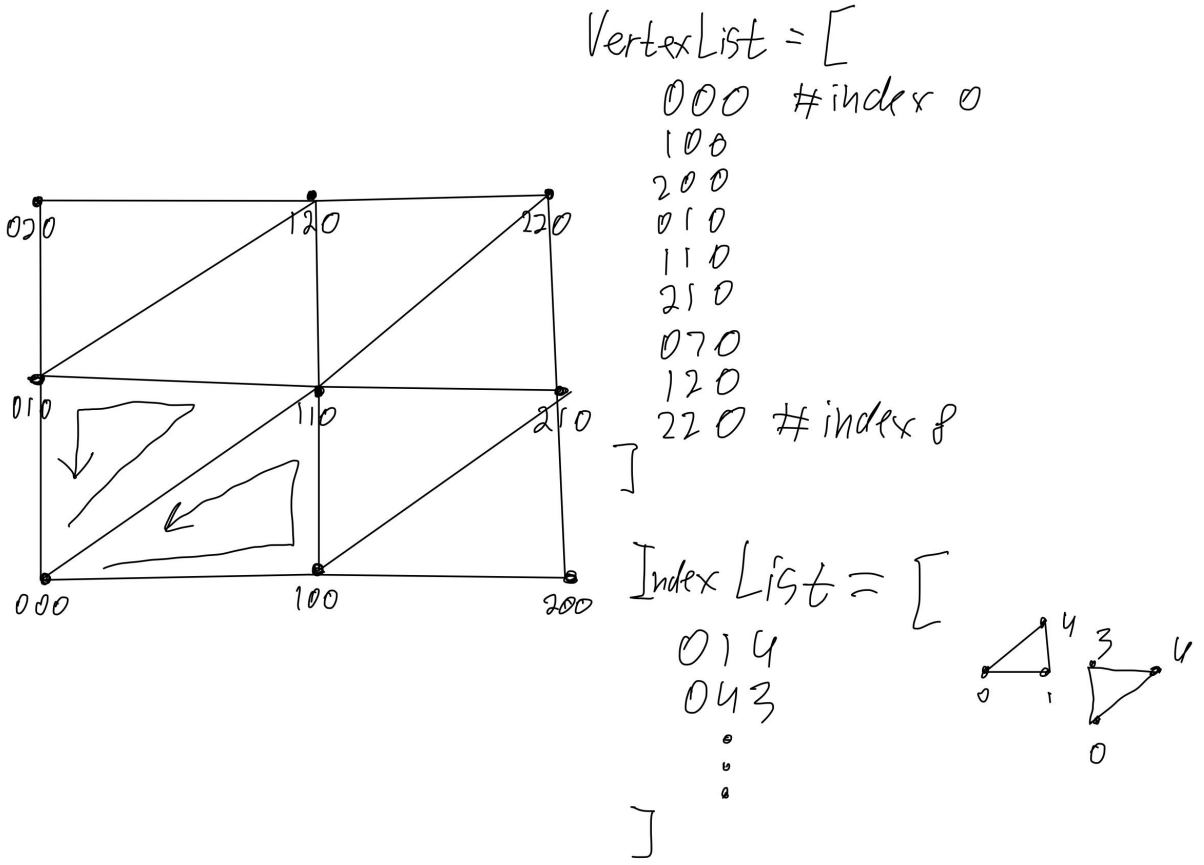
Meshgen.rs

This file is responsible for all mesh generation. Firstly a simple geN-skybox function. It generates a simple UV sphere of a very big radius and has its indices reversed, so that the texture could be view from inside. This function is used while generating the skybox in the scene.rs.

The gen_ter_mesh function generates a mesh given its integer size, position and layer parameters for the noise algorithm. Initially a blank mesh is created through Bevy's PrimitiveTopology:TriangleList. In order to generate a basic mesh in bevy, we will need the following:

- Vertex array - those are the building blocks of the mesh, initially a list of points will be generated with 0 height like (0, 0, 0), (1, 0, 0), (2, 0, 0) and so on until (N, M, 0)
- Index list - In order for the geometry to display properly mesh should be described in its simplest primitives - triangles. Index list describes each individual triangle out of 3 vertices, so every 3 values in the index list describe a single triangle by pointing to 3 vertices in the vertex list.
- Normal list - a vector list that describes the direction at which triangles point to. This property will be crucial for the light and colour calculation later down the line. For now it will be defaulted to (0, 0, 0) as now mesh is a simple plane.

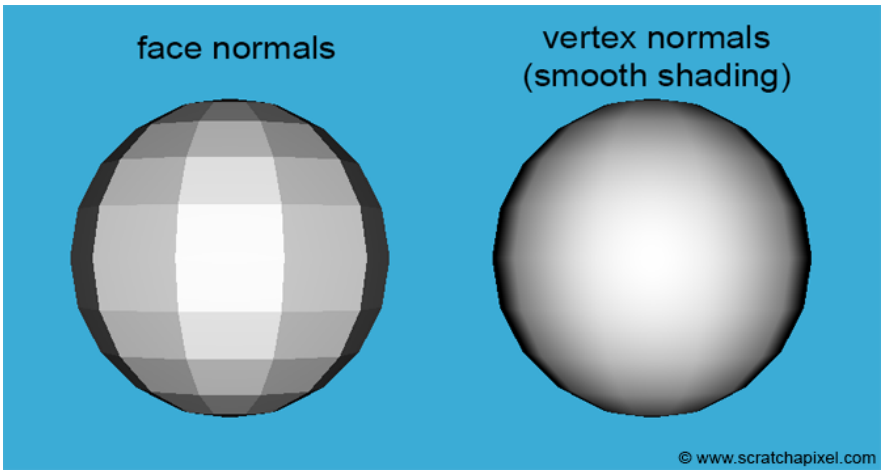
First blank mesh is created if TriangleList. Then a positions array is generated using the perlin algorithm, then indices are created, by cycling through the whole mesh and splitting the square polygons which vertices form into little triangles, those triangles are the most primitive structure in 3D graphics, so the only thing that is needed to describe it is 3 points and index array, the index array has indices which point to the position of vertex which should form a connection between them, conventionally in 3D graphics, when facing the triangle its indices should be turned couter-clockwise, that means that triangle is facing the right way.



Indices are generated by the separate function generate_indecies, it takes in only one argument size, which describes the size of the grid. First a vector of integers is generated; those are indices. A cycle is run which cycles a square windows through the vertex structure, each square which has 4 points is divided into 2 triangles, each triangle point could be determined by the offset of the window, the index in the next column for example will have an index offset of 1, but a vertex on another column will be exactly N points away, where N is the width if the grid, so the described cycle inserts two triangles which form a square by plugin in the triplets of indices, then the vertex is incremented by one to shift the window to the next in line and incremented by one one more time if column was switched, triangle count is increased by 6 as exactly 2 triangles with 3 points each were recorded, so next time the indices are inserted they are not overwritten. The function returns a final array of indices, each triplet of them describes a single triangle in the grind.

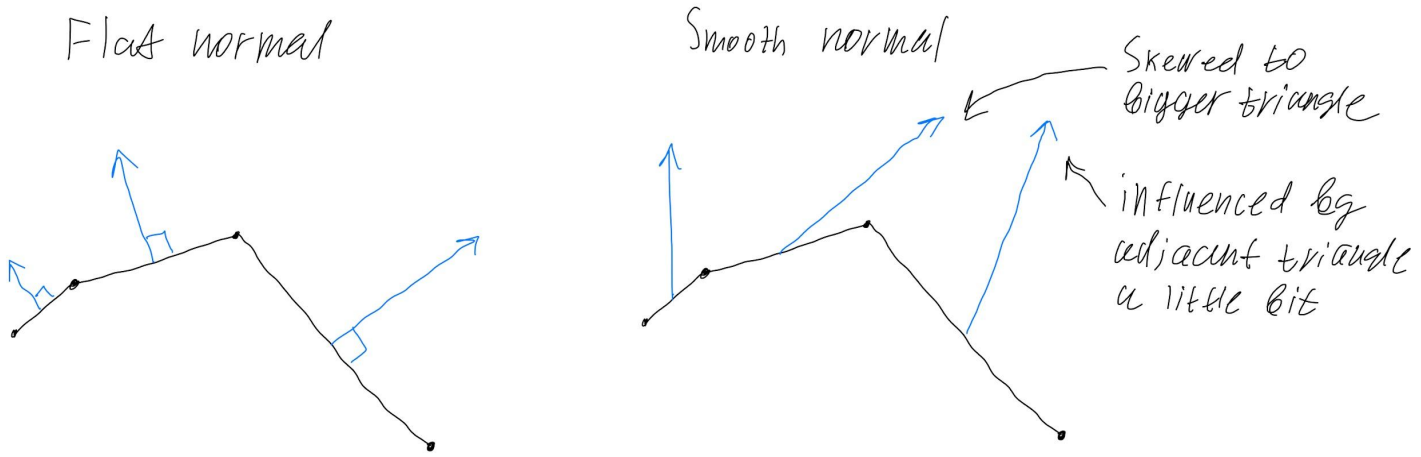
```
let mut idx: Vec<u32> = vec![0 as u32; (size.0 * size.1 * 6) as usize];

let mut vert: u32 = 0;
let mut tris: usize = 0;

for _ in 0..size.0 {
        for _ in 0..size.1 {
                idx[tris + 0] = vert;
                idx[tris + 1] = vert + 1;
                idx[tris + 2] = vert + size.1 + 1;

                idx[tris + 3] = vert + size.1 + 2;
                idx[tris + 4] = vert + size.1 + 1;
                idx[tris + 5] = vert + 1;

                vert += 1;
                tris += 6;
        }

        vert += 1
}
```

Now normals could be recalculated. Every mesh could have its normals calculated “flat” and “smooth”. Flat normals are calculated by pointing normal of two triangles inside a face perpendicular to that face. This gives flat normals a more square look. Smooth normals however, are calculated by taking the average perpendicular normal of 3 adjacent triangles proportionally to their area, so that a big triangle will influence its neighbour normal calculation more than a small triangle. This way the mesh comes to be more smooth.



The generate_smooth_normals function takes in the size of the grind and a vector of positions. Firstly an empty vector of arrays is initialised this will contain the direction vectors for each individual triangle in the mesh. Then the for cycle goes through every single vertex in the mesh until the end is reached. Unless on the edge, each point is connected to at least 4 more points around it, those points form 4 triangles. According to the offset of those vertices the triangles will face different directions so the normal should be averaged among those triangles’ normals. The final normal of the vertex is calculated by averaging out the normals to each individual triangle around that point. One could simply calculate the normals to those 4 triangles and average

them to obtain a simple normal, however it does not look good. Instead the weighted average should be calculated according to the area of the triangles formed with those 4 points, so that the triangles that have a bigger area have more influence on the direction of the final normal.



Now normalised 4 points could be used to calculate 4 normals and 4 areas. To perform those calculations mathop.rs is used. Cross product is applied for each pair of neighbouring points to find the normal to each triangle and an area is found using Heron's formula which, given 3 lengths, calculates the area of the triangle. So now having the area of a triangle as well as each individual normal a weighted average could be found by multiplying each direction vector with area, summing those 4 normals together and dividing them by the total area of 4 triangles, this way a weighted average is found and normals are smooth, which makes the final mesh look better.

Now that the mesh structure is completely defined a final touch of coloured vertices could be applied. The y position of each vertex could be used to give it a unique colour, for example in the lower areas like coasts there is likely to be sand, so below a certain point yellow colour could be applied to the vertex, or for higher points like peak of mountains white colour could be used to simulate snow and so on. So a vector of colour points is generated, each colour point consists of a red, green and blue channel as well as an alpha channel which determines the transparency. So external function color_height is used to determine the colour of each individual point in the mesh and a colour is recorded into a separate vector, this way when the final mesh is rendered the Bevy engine can use those colour points to colour the mesh procedurally. The colour height function is a predefined colour ramp which takes in a single y height value and returns the colour point, the colour bands are predefined with sand, grass, stone and snow level. WIth colour points generated the single chunk mesh is done.

Function name	Parameters	Output data type	Description
gen_skybox	N/A	Bevy Mesh	Generates a UV Sphere and inverts its indecies
inv_idx	1. Vector of u32 indices	Vector of u32 indices	Inverts the order of triplets in the indices vector, to invert the normal direction of the triangle
gen_ter_mesh	1. Integer tuple Size - (N, M) 2. F32 position – (X, Y) 3. Layers vector of i32	Bevy Mesh	Generates a simple grid of 3D points and applies Perlin noise to them. Then the normals are recalculated and vertices are colored according to

	triplets		their height.
gen_idx	1. Integer tuple Size - (N, M)	Vector of u32	Generates indices of a plane mesh with resolution of N by M points
gen_normals	1. Integer tuple Size - (N, M) 2. Vector of f32 3D points- A	Vector of f32 3D direction vectors	Calculates flat normals to each point in vector A by taking a perpendicular to the triangle formed by this point
gen_smooth_normals	3. Integer tuple Size - (N, M) 4. Vector of f32 3D points- A	Vector of f32 3D direction vectors	Calculates smooth normals to each point in vector A by taking a weighted average of 4 adjacent triangles to this point
color_height	1. f32 3D point - (X, Y, Z)	RGBA Colour	Colours the point according to its height value, low is water, mid is sand, higher is stone and the highest is white snow.

In depth design

Initialisation

First comes the initialisation. Bevy app is initialised with `App::new()`.
Then general system settings are set with `WindowDescriptor` resource.

```
.insert_resource(WindowDescriptor {
            height: 800.0,
            width: 600.0,
            title: "Terrust 1.4`".to_string(),
            ..default()
})
```

Then the UI system is initialised by inserting a Bevy eGUI plugin, adding the described ui_system in ui.rs, inserting the UiState resource which stores all of the sliders and buttons values that user tweaks and UI_state_defaults, which runs runs only once at the start of the program and set the defaults for the UiState.

```
.add_plugin(EguiPlugin)
.add_system(ui_system)
.init_resource::<UiState>()
.add_startup_system(ui_state_defaults)
```

Then a ChunksResource is initialised with

```
.init_resource::<ChunksResource>()
```

After that a user camera is added with an external library, the movement speed and mouse sensitivity are set accordingly.

```
.add_plugin(PlayerPlugin)
.insert_resource(MovementSettings {
    sensitivity: 0.0002,
    speed: 1400.0,
})
```

Then startup systems for scenes and chunks are added to create the world.

```
.add_startup_system(basic_scene)
.add_startup_system(chunks_init)
```

And finally chunk systems are setup for update and flush

```
.add_system(chunks_update_sys)
.add_system(chunk_timed_flush)
```

Chunk object design

As described the ChunksResource has only one field which (chunks) which is a vector of Chunks objects instances.

```
pub struct ChunksResource {
        pub chunks: Vec<Chunk>
}
```

Chunks themselves have two fields, `mesh`, which stores the entirety of the mesh and is used when the chunk is exported and `display` which is a flag for the `chunks_update_sys` which identifies which chunks are already generated and which are missing and need to be created.

```
pub struct Chunk {
        pub mesh: Mesh,
        pub display: bool,
}
```

Mesh indicator is a blank structure with no fields in it, used only for marking the chunk objects in the game world. It is used in flush method of `ChunksResource`

```
pub fn flush (
      ...
      mut query: Query<Entity, With<MeshIndicator>>,
      ...
) {
      query.for_each(|entity| {
            commands.entity(entity).despawn();
      });
```

```
...}
```

The .save method of ChusResource takes in a integer coordinates and converts them into vector index, knowing the vertical and horizontal size of the terrain in chunks (Global constant `CHUNK_MAX_NUM`) it is possible to calculate the index of a chunk with coordinates (x, Y) in a vector chunks by using multiplication

```
...
let index_in_vector = x_position + y_position * CHUNK_MAX_NUM_ALONG_Y;
let extracted_chunk = ChunksResource.chunks[index_in_vector];
...
```

For .save to be compatible with exporting library indices are needed to be reformatted from a continuous Vec<i32> to Vec<[i32; 3]> a vector of triplets of i32.

```
let indices = extracted_chunk.mesh.indices();
let corrected_indecies = vec_chunk_triplets(indices);

let vertices = extracted_chunk.mesh.attribute(Mesh::ATTRIBUTE_POSITION);
let triangle_mesh = TriMesh::new(vertices.to_vec(), corrected_indecies);
```

And finally export the mesh using the meshx library

```
meshx::io::save_trimesh(&triangle_mesh, file_name);
```

Meshgen.rs

Arguably one if the most complex files to understand conceptually.
The `gen_ter_mesh` produces a mesh for the chunk given its size (N, M), position and layers settings. First a blank mesh is initialised from the Bevy's `PrimitiveTopology` module – `TriangleList.`

```
let mut mesh = Mesh::new(bevy::render::mesh::PrimitiveTopology::TriangleList);
```

After the noise is generated for the chunk – `let pos = gen_perlin(size, pos, layers);`
Mesh is updated with new attributes, such as the index list of N by M points from a generated grid and vertices cloned from the perlin noise function output.

```
mesh.set_indices(Some(bevy::render::mesh::Indices::U32(gen_idx(size))));
mesh.insert_attribute(Mesh::ATTRIBUTE_POSITION, pos.clone());
```

After the points are inserted successfully, smooth normals are calculated from the same `pos.clone()`

```
mesh.insert_attribute(Mesh::ATTRIBUTE_NORMAL, generate_smooth_normals(size,
pos.clone()));
```

And finally the last attribute is inserted `ATTRIBUTE_COLOR` which allows for the individual vertices to be procedurally coloured according to their height in the world. Positions which is a vector of 3D points is iterated

through, for each 3D point a function is applied `color_height` which returns a pre-defined colour according to the points vertical position in space, the output of the iterator is collected and converted to vector of arrays of 4 float32.

```
mesh.attribute(Mesh::ATTRIBUTE_POSITION) {
      let colors: Vec<[f32; 4]> = positions
            .iter()
            .map(|[x, y, z] | color_height(*x, *y, *z))
            .collect();
      mesh.insert_attribute(Mesh::ATTRIBUTE_COLOR, colors);
}
```

Namely the `color_height` function does not really need the x and z values, so they are dropped, a multiple if statements filter through numbers to clamp them to the pre-defined palette of colours which are recorded in a RGBA space.

```
fn color_height(_x: f32, y: f32, _: f32) -> [f32; 4] {
        let mut c = [0.5, 0.5, 0.5, 1.0]; //Standard gray

        if y > 80.0 {
                c = [1.0, 1.0, 1.0, 1.0] //White snow
        } else if y > 35.0 {
                c = [0.75, 0.75, 0.75, 1.0] //Grayish stone
        } else if y > 10.0 {
                c = [0.3921, 0.647, 0.2941, 1.0] //Green grass
        } else if y > 0.0 {
                c = [0.9764, 0.8941, 0.8196, 1.0] //Yellowish sand
        } else {
                c = [0.2627, 0.3725, 1.0, 1.0] //Blue water
        }

        return c
}
```

gen_idx function is one of the most complex generators in the project.

```
fn gen_idx(size: (u32, u32)) -> Vec<u32> {
    let mut idx: Vec<u32> = vec![0 as u32; (size.0 * size.1 * 6) as usize];

    let mut vert: u32 = 0;
    let mut tris: usize = 0;

    for _ in 0..size.0 {
        for _ in 0..size.1 {
            idx[tris + 0] = vert;
            idx[tris + 1] = vert + 1;
```

```
            idx[tris + 2] = vert + size.1 + 1;

            idx[tris + 3] = vert + size.1 + 2;
            idx[tris + 4] = vert + size.1 + 1;
            idx[tris + 5] = vert + 1;

            vert += 1;
            tris += 6;
        }
        vert += 1
    }
    return idx;
}
```

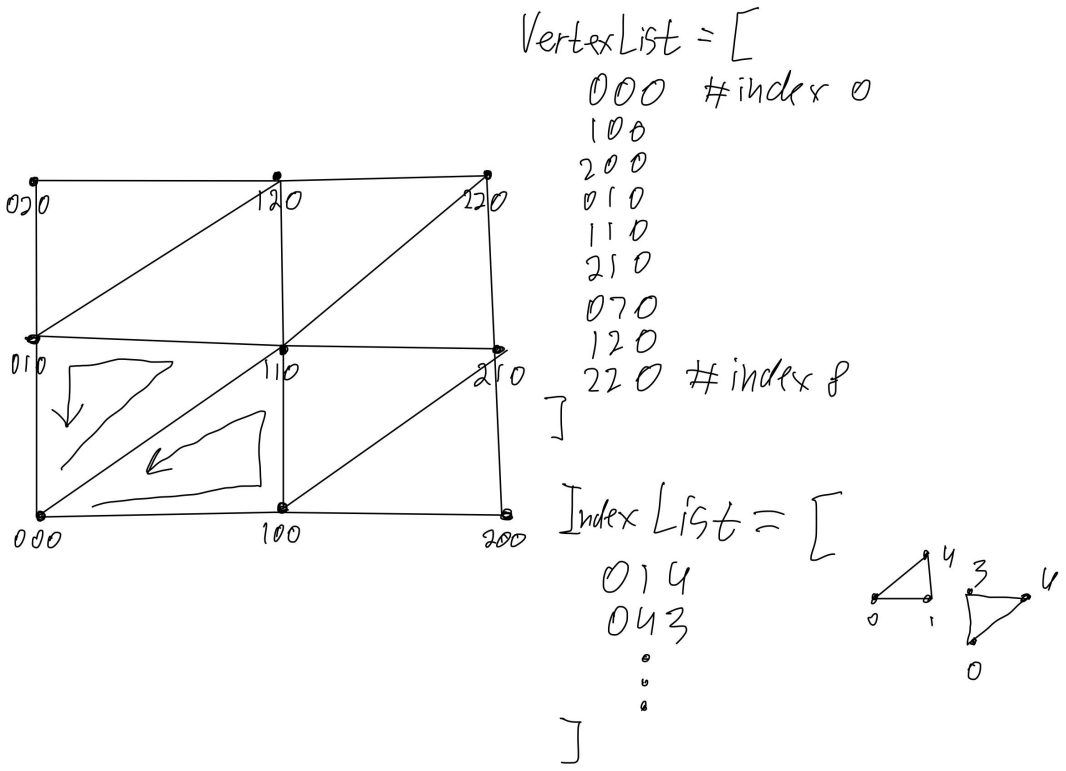
It takes in a u32 tuple (N, M) which is a desirable size of the grid and returns a vector of indices which are positioned exactly to make a plane out of this disconnected grid. The vector is fixed in size N*M*6 as there are exactly 2 triangles with 3 points each per one square dimension of the grid (quad).

```
fn gen_idx(size: (u32, u32)) -> Vec<u32> {
        let mut idx: Vec<u32> = vec![0 as u32; (size.0 * size.1 * 6) as usize];
...
```

The for loop iterates exactly N*M times, going through each point (iteration values are not needed so they are omitted → _)

```
for _ in 0..size.0 {
      for _ in 0..size.1 {
```

Each point will have 2 triangles it participates in. If the current point in space is 000 then the first triangle will be formed with a point above it and one more diagonally away from it, so the first indices that have to be described are of points 000, 010 and 110. But remember that the front size of the triangle is recorded counterclockwise, so the correct order will be 000, 110 and 010. In the supposed vertex list those points have indices 0, 1 and 4 so they are pushed back into the vector. Same process occurs for the adjacent triangle with points 000, 110, 100.



In the code the reference point is not 000 and is always changing so, knowing the dimensions of the grid it is possible to describe the point above and point diagonally opposite by only using value offsets.

If current vertex position in list is `vert` then vertex that is

- To the right of it is `vert + 1`
- Directly above is `vert + SIZE_Y + 1`

So the relative position of vertices that form a triangle could be recorded as (`vert, vert+1, vert+SIZE_Y+1)`. Same process applies to the adjacent triangle of the quad.

In the index list the described values will be recorded as such

```
idx[tris + 0] = vert;
idx[tris + 1] = vert + 1;
idx[tris + 2] = vert + size.1 + 1;

idx[tris + 3] = vert + size.1 + 2;
idx[tris + 4] = vert + size.1 + 1;
idx[tris + 5] = vert + 1;

vert += 1;
tris += 6;
```

Where `vert` is a variable that is incremented by one to go through each point; and `tris` is an offset for the vector that steps 6 indices at a time as exactly 2 triangles were described with 3 points each (2*3 = 6)

When the `for` cycle steps over an edge, `vert` is incremented by one to avoid misalignment.

```
for _ in 0..size.0 {
      for _ in 0..size.1 {
            ...
      }
      vert += 1
}
...
```

The `gen_smooth_normals` is the second most complex function in the project due to its meticulous *edge* cases.

```
fn gen_smooth_normals(size: (u32, u32), positions: Vec<[f32; 3]>) -> Vec<[f32; 3]> {
      let mut normals: Vec<[f32; 3]> = Vec::new();
      let mut o: [f32; 3];
      let mut a: [f32; 3];
      let mut b: [f32; 3];
      let mut c: [f32; 3];
      let mut d: [f32; 3];

      for i in 0..size.0+1 {
            for j in 0..size.1+1 {
                  o = positions[(i * (size.0 + 1) + j) as usize];

                  if i == 0 {
                        a = o
                  } else {
                        a = positions[((i - 1) * (size.0 + 1) + j) as usize];
                  }

                  if j == 0 {
                        d = o
                  } else {
                        d = positions[(i * (size.0 + 1) + j - 1) as usize];
                  }

                  if i == size.0 {
                        c = o
                  } else {
                        c = positions[((i + 1) * (size.0 + 1) + j) as usize];
                  }

                  if j == size.1 {
```

```
                    b = o
                } else {
                    b = positions[(i * (size.0 + 1) + j + 1) as usize];
                }

                normals.push(tr_wt_avg(o, a, b, c, d));
            }
        }

        return normals;
}
```

First a vector of f32 triplets is initialised for the final normals.

```
let mut normals: Vec<[f32; 3]> = Vec::new();
```

Then variable (mut) triplets are created to record the points which will be iterated later. The point `o` is an anchor point, which index will be used to address all the other points later. For a diagram refer to idx_gen function at page 40.

```
let mut o: [f32; 3]; //Anchor point
let mut a: [f32; 3];
let mut b: [f32; 3];
let mut c: [f32; 3];
let mut d: [f32; 3];
```

The function iterates through all of the points in the `positions` vector which is `size.0` horizontally and `size.1` vertically with 2 `for` cycles.
The `size.1+1` is added to include the top and right edges of the grid.

```
for i in 0..size.0+1 {
        for j in 0..size.1+1 {
...
```

Now relative offsets are used to record each point in the `oabcd` variables. The current point at (X,Y) is (i, j) which translates to `i * (size.0 + 1) + j` in the vector index format. So point o is

```
o = positions[(i * (size.0 + 1) + j) as usize];
```

The relative offset method in 4 directions is applied to obtain points a, b, c, d. For the Relative offset method refer to the idx_gen on page 40. Same goes for this one but instead of 2 directions, there are 4 directions and an offset could be negative as well to address points to the left and down to the `o`.

There are some *edge* cases to be accounted for, for example when i or j is equal to 0 or size.0 and 0 or size.1 respectively it means that the point is on the edge and there are no neighbouring points to the direction of the edge, so for each case an if check was added to fix that and replace the missing point with anchor point.

```
if i == size.0 {
      c = o
} else {
      c = positions[((i + 1) * (size.0 + 1) + j) as usize]; //Relative offset method
}
```

When the points `oabcd` are obtained it is time to calculate the smooth normal with `tr_wt_avg` function and add it to the vector.

```
normals.push(tr_wt_avg(o, a, b, c, d));
```

Mathop.rs

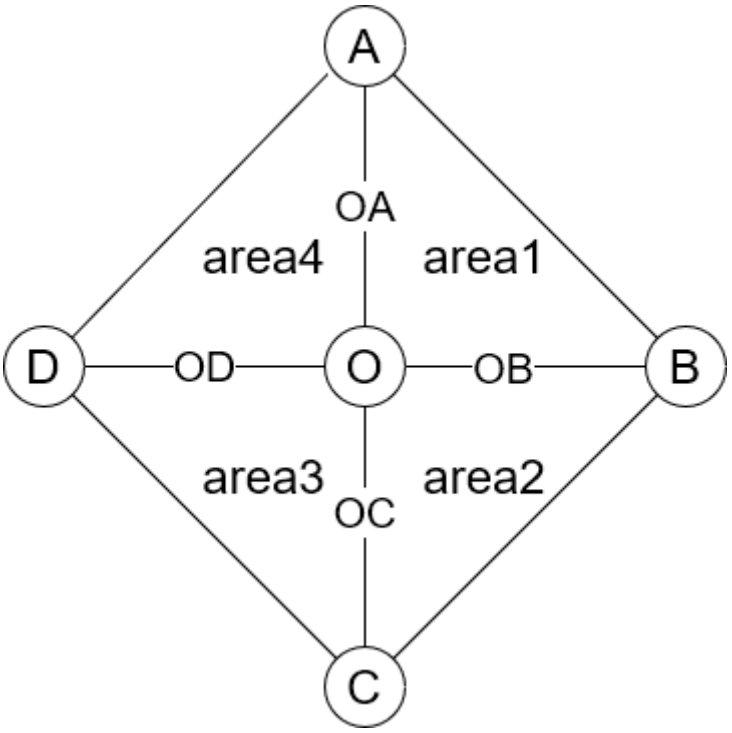
As a simple example an arbitrary vector was fed into `vec_chunk_triplets` to test if it works

```
>> vec_chunk_triplets(Vec<[1, 2, 3, 4, 5, 6]>)
Vec<[i32; 3]> = Vec<[(1, 2, 3), (4, 5, 6)]>
```

Same was performed on mathop.rs functions such as `cross_prod` and `vec_mag` to confirm the validity of the function outputs.

```
>> cross_prod([2.0, -3.0, 1.0], [4.0, -1.0, 5.0])
[f32; 3] = [-14.0, -6.0, 10.0]

>>vec_mag([2.0, -4.0, 6.0])
f32 = 7.48331477... // sqrt(56)
```

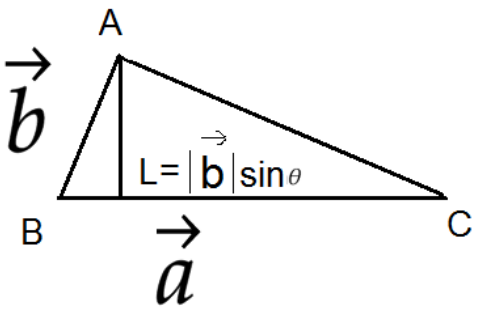
The tr_wt_avg produces a smooth normal for vertex O by using its adjacent points A, B, C, D (A~D)



First O is subtracted from A~D to anchor all of them to the origin O.

```
let oa = vec_sub(o, a);
let ob = vec_sub(o, b);
let oc = vec_sub(o, c);
let od = vec_sub(o, d);
```

Now 4 direction vectors are known for all 4 triangles, this allows the usage of the cross product of two vectors to calculate the area of the formed triangles, which is precisely what `tr_area` function is doing.



$$\text{area of } \triangle = \frac{1}{2} \times |\vec{a}| \times |\vec{b}| \times \sin\theta$$

$$= \frac{1}{2} \times [\vec{a} \times \vec{b}]$$

```
fn tr_area(a: [f32; 3], b: [f32; 3]) -> f32 {
```

```
            return 0.5*(
                        (a[1]*b[2] - a[2]*b[1]).powi(2) +
                        (a[2]*b[0] - a[0]*b[2]).powi(2) +
                        (a[0]*b[1] - a[1]*b[0]).powi(2)
                        ).powi(1/2);
}

...

let area1 = tr_area(oa, ob);
let area2 = tr_area(ob, oc);
let area3 = tr_area(oc, od);
let area4 = tr_area(od, oa);
```

After the area of each triangle is calculated the A~D are normalised

```
let a = vec_norm(oa);
let b = vec_norm(ob);
let c = vec_norm(oc);
let d = vec_norm(od);
```

After that a normal to each triangle is found by again using a cross product on each of the adjacent pairs

```
let tr1 = cross_prod(a, b);
let tr2 = cross_prod(b, c);
let tr3 = cross_prod(c, d);
let tr4 = cross_prod(d, a);
```

And finally the normals are scaled by the area of their corresponding triangles and added up together into final_vec, which is then scaled down to the sum of all areas to make it an average weighted normal.

```
final_vec = vec_add(final_vec, vec_scale(tr1, area1));
final_vec = vec_add(final_vec, vec_scale(tr2, area2));
final_vec = vec_add(final_vec, vec_scale(tr3, area3));
final_vec = vec_add(final_vec, vec_scale(tr4, area4));
final_vec = vec_scale(final_vec, 1.0/(area1 + area2 + area3 + area4));

return final_vec
```

Systems.rs

The chunk update system is the single most important system in the file. It is responsible for generation of the chunks near the camera of the viewport.

First a camera is found in the scene using a Bevy query and its position coordinates are taken.

```
...
mut camera: Query<&mut Transform, With<Camera>>,
...
let cam_position: (f32, f32) = (cam.translation[0], cam.translation[2]);
```

This f32 position is then converted to integer value according to the universal world scale. Of course edge cases are considered.

```
let mut int_coordinates: (u32, u32) = (0, 0);

// <0 edge cases
if cam_position.0 < CHUNK_MAX_NUM.0 as f32 * SCALE {
      int_coordinates.0 = (cam_position.0 / SCALE) as u32;
}
if cam_position.1 < CHUNK_MAX_NUM.1 as f32 * SCALE {
      int_coordinates.1 = (cam_position.1 / SCALE) as u32
}
```

After the integer coordinates are obtained a for cycle iterates through the coordinates which are within the render distance.

```
for near_int_coordinates in chunks_load_near(int_coordinates,
ui_state.render_distance) {
```

The obtained coordinate is converted to index value for vector

```
let idx = (near_int_coordinates.0 + near_int_coordinates.1 * CHUNK_MAX_NUM.1 as i32)
as usize;
```

And the chunk is checked, if it is not yet loaded a mesh is generated, positioned according to the coordinates used and the chunk is inserted into the vector.

```
if chunks.chunks[idx].display == false { // If not loaded generate it
      let ChunkMesh = meshgen::gen_ter_mesh(CHUNK_RES, (near_int_coordinates.0 as f32
* CHUNK_RES.0 as f32, near_int_coordinates.1 as f32 * CHUNK_RES.1 as f32),
ui_state.layers.clone());
      commands.spawn_bundle(PbrBundle {
            mesh: meshes.add(ChunkMesh.clone()),
            transform: Transform::from_scale(Vec3::splat(SCALE / CHUNK_RES.0 as
      f32)).with_translation(Vec3{x: near_int_coordinates.0 as f32 * SCALE, y: 0.0,
      z: near_int_coordinates.1 as f32 * SCALE}),
            material: materials.add(StandardMaterial {
            base_color: Color::rgba(1.0, 1.0, 1.0, 0.0),
            perceptual_roughness: 1.0,
            metallic: 0.0,
```

```
      ..default()}),
..Default::default()
      }).insert(MeshIndicator{}); // Mesh indicator for marking to use in despawn
later
      chunks.chunks[idx].display = true;
      chunks.chunks[idx].mesh = ChunkMesh; // Saved for export function
}
```

Terrain.rs

The Perlin layering algorithm works by going through multiple frequencies of perlin noise with different height multipliers, so that ofr example a less frequent noise that resembles pits and mountains is more influential rather than high frequency noise that adds those small details.
For each point in the passed array a layered perlin point is generated according to the parameters given in the layer vector.

```
...
for x in 0..size.0+1 {
      for z in 0..size.1+1 {
            for l in layers.clone() {
                  if l[2] == 1 {
                        finally += perlin.get([
                              ((x as f64 + offset.0 as f64) * PERLIN_FREQ * l[0] as
                              f64),
                              ((z as f64 + offset.1 as f64) * PERLIN_FREQ * l[0] as
                              f64)]) * PERLIN_HEIGHT_SCALE * l[1] as f64;
                        }}
...
```

For each point a layer vector is checked for parameters, those parameters are used to generate the perlin noise with different frequencies and amplitudes, when added up it creates an illusion of detailed terrain.

Testing

Overview of test strategy

The testing will consist of a general overview of the defined requirements, a video demonstration of the project and timestamps pointing out the completeness of requirements.

Overall requirements

- Final application is free, open source and can be compile for Windows and Linux
- Mesh must me algorithmically generated

- The generator function must be adjustable
- App must export high resolution mesh
- The preview must be smooth and feature user controls for comfortable preview
- Export terrain segment to the OBJ file
- Render distance must be adjustable
- After generation parameters are changed user should be able to regenerate the mesh
- Mesh is shaded smooth for comfortable preview
- Sky as background
- Mesh is coloured procedurally

Test Plan

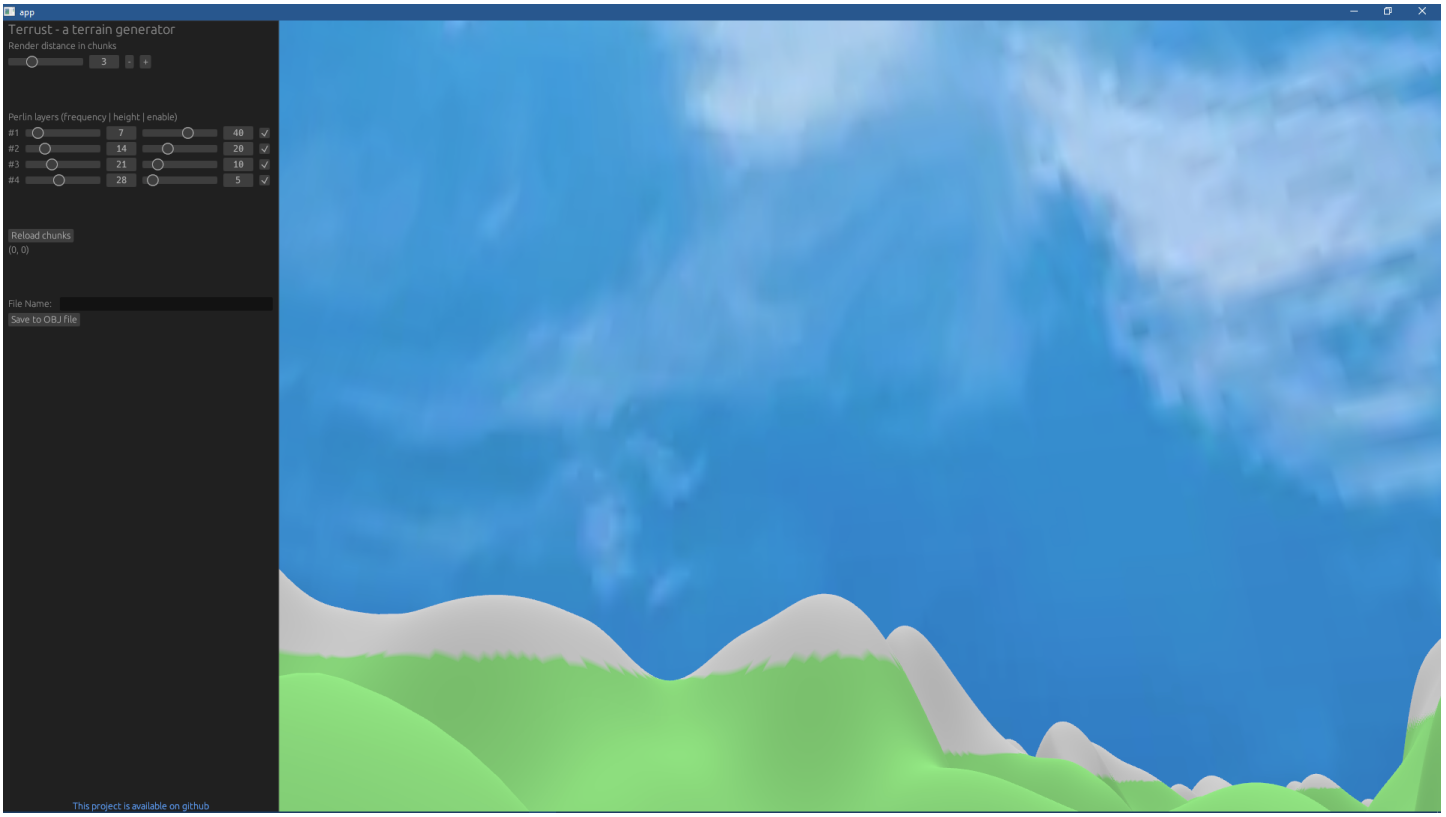
I will be using my home laptop to commence testing. It has Windows 10 and Arch Linux installed on it. I will run the compiled executable of Terrust, generate a mesh, export it to a file and attempt to open it with a 3rd party 3D editing software Blender.

Evidence

For the evidence of commenced tests please see the video attachment as well as the screenshots.

Video is available on YouTube (see the moderator cover sheet for link)

Screenshot 1



Screenshot 2

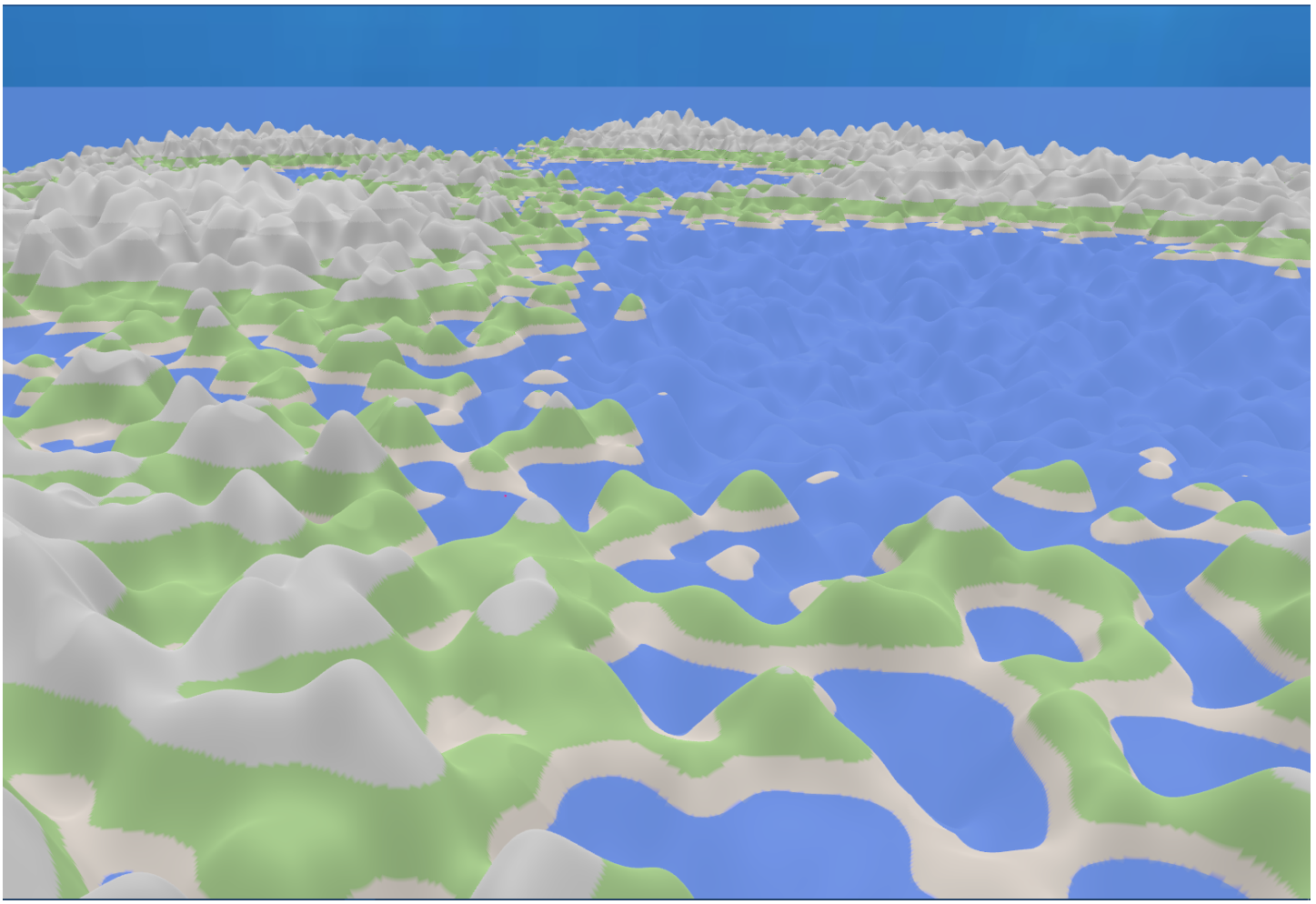


Table of objectives and their timecodes in the video.

Objective	SUCCESS?	EVIDENCE
Application is free	PASS	4:25
Source code available online	PASS	4:25
Terrust runs on Linux	PASS	0:10
Terrust runs on Windows	PASS	2:27
Window with a 3D preview of the generated mesh	PASS	0:20
The viewport camera can be controllable by the user	PASS	0:20
Preview is smooth and does not stutter when camera is moved	PARTIAL	0:20
Render distance must be adjustable	PASS	2:48
After parameters are adjusted user should be able to regenerate the mesh	PASS	3:01
Shade the mesh smooth for comfortable preview of the mesh	PASS	3:15
Mesh is coloured according to vertex height	PASS	Screenshot 2

Sky as a background	PASS	Screenshot 1
Mesh is algorithmically generated	PASS	0:30
The generation parameters are adjustable	PASS	0:35
App must export high resolution mesh	PASS	2:16
Save button exports mesh to OBJ file	PASS	1:17

Evaluation

Overall evaluation

In general the software produced has met almost all of the criterias that were defined at the start. The app is free and open source available online, it runs smoothly ,however there is a visible freezes visible for a split second when the new chunks are generated. (video: 0:30) The combined perlin layers produce a terrain like mesh that is exportable to the OBJ format. In the video at 2:16 the view mode was switched to wireframe to show the mesh resolution, however due to low video bitrate some moving parts were smudged.

Evaluation against objectives

1. Software type
    a. The projects is available for free to download from online repository
    b. Project is published online on github with its source code
    c. It is possible to build the app for Windows and Linux, however I have not tried other OS like MacOS
2. Interface
    a. App has a 3D camera and mesh is seen inside of it
    b. Controls are possible with WASD layout, possibly automatic camera movement could be added in the future
    c. The overall experience during the runtime of the program is quite smooth, but with increasing render distance the app starts to take more and more time to generate the chunks. Possibly a benchmark could be performed to identify the slowest parts of the code, most likely the mesh update function is the culprit and it needs to be optimised.
    d. The adjustable render distance does allow for a big mesh preview, however as mentioned on increasing scale programs starts stuttering, one solution to that could be adaptive level of detail, which will decrease the visual resolution of the chunks that are too far away from camera, this way an increase in performance will be achieved with no visible loss to the quality of the mesh. As well as that a RTIN optimisation could be used on a square mesh, by reducing the number of vertices in the places where there is not that much variation in detail, such as flat segments.
    e. Render distance is adjustable, however it should be noted that at the moment it is limited at 32 chunks, which is a current limit of the program that can work stable, in the future when optimisations are made to update system and mesh generation function

maximum distance could be increased

f. After user has adjusted the sliders, they can press an update button, however sometimes it takes too long as the whole mesh should be regenerated, this could be possibly fixed with adaptive update system as discussed previously
g. The final mesh is shaded smooth and looks good.
h. The final mesh is coloured according to the vertex colour, however at a closer inspection to the areas where elevation change is too big, individual color patches start to be seen, this could possibly be fixed by increasing the resolution of the mesh or using other colouring methods.
i. Sky is indeed appears at a background, however the resolution of the texture appears to be quite low, in the future I could use higher resolution texture

3. Mesh
    a. Mesh is generated algorithmically, however only one generation algorithm is available at the moment – Perlin noise, in the future I would add more noise algorithms as well as introduce features like soil degradation simulation and caves generator. Additionally the mesh generating function could be cached when it is called with the same parameters, this will increase the speed of generation function and update systems. Moreover Rust allows for concurrent code architecture, so the generation of multiple chunks at the same time could be possible if the user's hardware allows for multiple threads.
    b. The generation parameters for perlin noise are available, but in the future I would increase the number of layers and make it a dynamic number instead of fixed like now 4.
    c. The mesh exported is of high resolution, possibly in the future I could add an option to change the maximum resolution of the mesh suitable for the user needs.
    d. The final exported file is of OBJ type that is compatible with most of the 3D software and is possible to export.

User feedback

I have presented the video demonstration of Terrust to my interviewee ■■■ that I questioned about the initial objectives. He has commented on good 3D preview and smooth shading, however added that software lacks in generation methods and it would be beneficial to add more options. The export functionality was commented to be sufficient enough, as OBJ format is generally recognized by the majority of the 3D editing software. And finally the interviewee commented that it would be good to have some overlays in the 3D preview like CPU and RAM usage as well as mesh information like resolution and number of triangles.

Reflection on user feedback

Indeed the software would be better off if more options were added in terms of mesh generation as at the moment the functions are quite limited. The 3D viewport could be modified to have an informative overlay like resource usage and mesh stats, that would require a slight modification to the ui.rs structure though. In the future I could increase the compatibility of the export by introducing other export formats like CAD or glTf.

Improvements / Extensions

- Add more options to mesh generation methods (soil degradation, valleys, caves)
- Include informative overlay in the 3D preview with resources usage and mesh information
- Expand on mesh export options (CAD, glTF)

- Optimise the mesh generation function by using function caching
- Chunk update system could be made parallel to utilise the multiple cores of host machine
- Chunks could be compressed at a points where there are not that much details using a RTIN reduction algorithm
- Adaptive level of details could be introduced to enable even bigger grid display

Implementation Cover Sheet

Skills Sheet

File	Skills	Page Number(s)
main.rs	Use of ECS in project	
chunk.rs	● OOP ● use external library for export ● complex mesh generation algorithms ● use of coordinate geometry for chunks positions	
mathop.rs	● geometry calculations ● area of a triangle with cross product ● vector calculations ○ subtraction ○ normalisation ○ cross product ○ inversion ○ scaling)	
meshgen.rs	● procedural mesh generation ○ vertices grid ○ indices ○ smooth normals ● procedural generation of mesh through use of relative offsets and indices manipulations.	
scene.rs	● scripted world building ● simple shading technique (alpha blend)	
terrain.rs	● procedural generation of offset points with Perlin noise layering	
systems.rs	● update chunk generating system ● use of coordinate geometry ● ECS resources manipulation	
ui.rs	● User interface design	

Implementation

Source Code on GitHub (https://github.com/idf3da/Terrust)

main.rs

```
mod terrain;
mod meshgen;
mod mathop;
mod scene;
mod systems;
mod chunk;
mod ui;

use crate::scene::*;
use crate::systems::*;
use crate::ui::*;
use crate::chunk::*;

use bevy_egui::egui;
use bevy::prelude::*;
use bevy_flycam::{PlayerPlugin, MovementSettings};
use bevy_egui::{EguiPlugin};

const S: u32 = 8;
const CHUNK_RES: (u32, u32) = (2 << S, 2 << S);
const CHUNK_MAX_NUM: (u32, u32) = (64, 64);
const SCALE: f32 = (S * S * 100) as f32;
const PERLIN_HEIGHT_SCALE: f64 = 2.0;
const PERLIN_FREQ: f64 = 0.0009765625;

fn main() {
        println!("Starting Terrust!");
        App::new()
                //Initialise bevy app
                .add_plugins(DefaultPlugins)
                .insert_resource(WindowDescriptor {
                        height: 800.0,
                        width: 600.0,
                        title: "Terrust 1.4`".to_string(),
                        ..default()
                })

                //Setup the UI system
                .add_plugin(EguiPlugin)
```

```
                .add_system(ui_system)
                .init_resource::<UiState>()
                .add_startup_system(ui_state_defaults)

                .init_resource::<ChunksResource>()

                //Add movement camera
                .add_plugin(PlayerPlugin)
                .insert_resource(MovementSettings {
                        sensitivity: 0.0002,
                        speed: 1400.0,
                })

                //Create a world
                .add_startup_system(basic_scene)
                .add_startup_system(chunks_init)

                //Systems
                .add_system(chunks_update_sys)
                .add_system(chunk_timed_flush)
                .run();
}
```

scene.rs

```
use bevy::prelude::*;
use crate::meshgen;

pub fn basic_scene(
    mut commands: Commands,
    mut meshes: ResMut<Assets<Mesh>>,
    asset_server: Res<AssetServer>,
    mut materials: ResMut<Assets<StandardMaterial>>
) {
    // Global light
    commands.spawn_bundle(DirectionalLightBundle {
        directional_light: DirectionalLight {
            illuminance: 30000.0,
            ..default()
        },
        transform: Transform::from_xyz(0.0, 0.0, 0.0)
            .with_rotation(Quat::from_rotation_x(-3.14 / 2.0)),
        ..default()
    });
```

```
    let skybox_box = meshes.add(meshgen::gen_skybox());
    let texture_handle = asset_server.load("textures/sky.png");

    // Skybox material
    let material_handle = materials.add(StandardMaterial {
        base_color_texture: Some(texture_handle.clone()),
        alpha_mode: AlphaMode::Blend,
        unlit: true, // Does not cast shadows
        ..default()
    });

    // Skybox
    commands.spawn_bundle(PbrBundle {
        mesh: skybox_box.clone(),
        transform: Transform::from_xyz(0.0, 0.0,
0.0).with_rotation(Quat::from_rotation_x(std::f32::consts::PI / 2.0)),
        material: material_handle,
        ..default()
    });

    // Water plane
    commands.spawn_bundle(PbrBundle {
        mesh: meshes.add(Mesh::from(shape::Plane {
            size: 10000000.0,
        })),
        material: materials.add(StandardMaterial {
            base_color: Color::rgba(0.35, 0.55, 0.85, 0.733),
            alpha_mode: AlphaMode::Blend, // Prevents clipping outside of terrain
area
            unlit: true,
            ..default()
        }),
        transform: Transform::from_xyz(0.0, 0.0, 0.0),
        ..default()
    });

}
```

chunk.rs

```
use bevy::prelude::*;

use crate::{
```

```
        CHUNK_MAX_NUM,
};

use bevy::render::mesh::VertexAttributeValues;
use bevy::render::mesh::Indices;
use meshx::TriMesh;

pub struct Chunk {
        pub mesh: Mesh,
        pub display: bool,
}

#[derive(Default)]
pub struct ChunksResource {
        pub chunks: Vec<Chunk>
}

#[derive(Component)]
pub struct MeshIndicator {}

impl ChunksResource {
        // Despawn all chunks that exist in the world
        pub fn flush (
                &mut self,
                mut query: Query<Entity, With<MeshIndicator>>,
                mut commands: Commands,
        ) {
                query.for_each(|entity| {
                        commands.entity(entity).despawn();
                });

                for i in 0..CHUNK_MAX_NUM.0 * CHUNK_MAX_NUM.1 {
                        self.chunks[i as usize].display = false;
                }

                print!("Flushed.")
        }

        pub fn save (
                &mut self,
                int_pos: (u32, u32),
                file_name: String
        ) {
                let chunk = &self.chunks[(int_pos.0 + int_pos.1 * CHUNK_MAX_NUM.1) as
usize];
```

```
                let mesh = &chunk.mesh;

                println!("Save initiated.");

                // Extract indecies in Vec format and chunk them into triplets
                // This is required by the standard meshx library index notation
                if let Some(VertexAttributeValues::Float32x3(positions)) =
mesh.attribute(Mesh::ATTRIBUTE_POSITION) {
                        if let Some(indices) = mesh.indices() {
                                let stuff = indices.to_owned();
                                if let Indices::U32(vector) = stuff {
                                        let trimesh =
TriMesh::new(positions.to_vec(), vec_chunk_triplets(vector));
                                        println!("{:?}",
meshx::io::save_trimesh(&trimesh, file_name).unwrap());
                                        println!("Save successfull.")
                                }

                        } else {
                                print!("Save failed Err #1.");
                        }
                } else {
                        print!("Save failed Err #2.");
                }

        }
}

// xxxxxx --> xxx, xxx
fn vec_chunk_triplets(vec: Vec<u32>) -> Vec<[usize; 3]> {
        let mut i = 0;
        let mut vec_res: Vec<[usize; 3]> = Vec::new();

        for _ in 0..(vec.len() as i32 / 3) {
                vec_res.push([vec[i as usize]  as usize, vec[(i + 1) as usize]  as
usize, vec[(i + 2) as usize]  as usize]);
                i += 3;
        }

        return vec_res


}

// At the start fills the world with blank chunks
```

```
pub fn chunks_init(mut chunks: ResMut<ChunksResource>) {
        chunks.chunks = Vec::new();
        for _ in 0..CHUNK_MAX_NUM.0 * CHUNK_MAX_NUM.1 {
                chunks.chunks.push(Chunk{
                        mesh:
Mesh::new(bevy::render::mesh::PrimitiveTopology::TriangleList),
                        display: false,
                });
        }
}
```

mathop.rs

```
// A, B vector

// A-B
pub fn vec_sub(a: [f32; 3], b: [f32; 3]) -> [f32; 3] {
        return [b[0] - a[0], b[1] - a[1], b[2] - a[2]];
}

// A+B
fn vec_add(a: [f32; 3], b: [f32; 3]) -> [f32; 3] {
        return [b[0] + a[0], b[1] + a[1], b[2] + a[2]];
}

// AxB
pub fn cross_prod (a: [f32; 3], b: [f32; 3]) -> [f32; 3] {
        return [a[1]*b[2] - a[2]*b[1], a[2]*b[0]-a[0]*b[2], a[0]*b[1]-a[1]*b[0]];
}

// Weighted average normal to O of 5 points with O being the center one.
pub fn tr_wt_avg(o: [f32; 3], a: [f32; 3], b: [f32; 3], c: [f32; 3], d: [f32; 3]) ->
[f32; 3] {
        let mut final_vec: [f32; 3] = [0.0, 0.0, 0.0];

        // Directional vectors
        let oa = vec_sub(o, a);
        let ob = vec_sub(o, b);
        let oc = vec_sub(o, c);
        let od = vec_sub(o, d);

        // First find area
        let area1 = tr_area(oa, ob);
        let area2 = tr_area(ob, oc);
        let area3 = tr_area(oc, od);
```

```
        let area4 = tr_area(od, oa);

        // Only then normalise
        let a = vec_norm(oa);
        let b = vec_norm(ob);
        let c = vec_norm(oc);
        let d = vec_norm(od);

        // Each normal to triangles
        let tr1 = cross_prod(a, b);
        let tr2 = cross_prod(b, c);
        let tr3 = cross_prod(c, d);
        let tr4 = cross_prod(d, a);

        // Sum their multiple
        final_vec = vec_add(final_vec, vec_scale(tr1, area1));
        final_vec = vec_add(final_vec, vec_scale(tr2, area2));
        final_vec = vec_add(final_vec, vec_scale(tr3, area3));
        final_vec = vec_add(final_vec, vec_scale(tr4, area4));

        // And divide by the sum of areas
        final_vec = vec_scale(final_vec, 1.0/(area1 + area2 + area3 + area4));

        return final_vec
}

// Area of a triangle with vector sides A and B
fn tr_area(a: [f32; 3], b: [f32; 3]) -> f32 {
        return 0.5*( (a[1]*b[2] - a[2]*b[1]).powi(2) + (a[2]*b[0] -
a[0]*b[2]).powi(2) + (a[0]*b[1] - a[1]*b[0]).powi(2) ).powi(1/2);
}

// |V|
fn vec_mag(v: [f32; 3]) -> f32 {
        return (v[0].powi(2) + v[1].powi(2) + v[2].powi(2)).powi(1/2).abs()
}

// A --> nA
pub fn vec_scale(v: [f32; 3], s: f32) -> [f32; 3] {
        let mut vs: [f32; 3] = [0.0, 0.0, 0.0];
        vs[0] = v[0] * s;
        vs[1] = v[1] * s;
        vs[2] = v[2] * s;
        return vs
}
```

```
pub fn inv_vec(v: Vec<[f32; 3]>) -> Vec<[f32; 3]> {
        let mut final_vec: Vec<[f32; 3]> = Vec::new();

        for v in v {
                final_vec.push(vec_scale(v, -1.0))
        }

        return final_vec
}

pub fn vec_norm(v: [f32; 3]) -> [f32; 3] {
        return vec_scale(v, vec_mag(v));
}
```

meshgen.rs

```
use bevy::prelude::*;
use bevy::render::mesh::VertexAttributeValues;

use crate::terrain::*;
use crate::mathop::*;


// SkyBOX is a UV sphere with inverted triangle direction
pub fn gen_skybox() -> bevy::prelude::Mesh {
   let mut skybox_mesh = Mesh::from(shape::UVSphere {
      radius: 1000000.0,
      sectors: 16,
      stacks: 8,
   });

   let idx_mesh = (*skybox_mesh.indices().unwrap()).iter().collect::<Vec<_>>();

skybox_mesh.set_indices(Some(bevy::render::mesh::Indices::U32(inv_idx(idx_mesh))));

   return skybox_mesh
}
```

```
pub fn gen_ter_mesh(size: (u32, u32), pos: (f32, f32), layers: Vec<[i32; 3]>) -> Mesh
{

        let mut mesh =
Mesh::new(bevy::render::mesh::PrimitiveTopology::TriangleList);
        let pos = gen_perlin(size, pos, layers);
        mesh.set_indices(Some(bevy::render::mesh::Indices::U32(gen_idx(size))));
        mesh.insert_attribute(Mesh::ATTRIBUTE_POSITION, pos.clone());
        // mesh.insert_attribute(Mesh::ATTRIBUTE_NORMAL, generate_normals(size,
pos.clone())); // Depriciated
        mesh.insert_attribute(Mesh::ATTRIBUTE_NORMAL, gen_smooth_normals(size,
pos.clone()));

        // Create a color for each vertex basing on its height
        if let Some(VertexAttributeValues::Float32x3(positions)) =
mesh.attribute(Mesh::ATTRIBUTE_POSITION) {
                let colors: Vec<[f32; 4]> = positions
         .iter()
         .map(|[x, y, z] | color_height(*x, *y, *z))
         .collect();
      mesh.insert_attribute(Mesh::ATTRIBUTE_COLOR, colors);
   }

        return mesh;
}

// Actually x and z are ignored, only y used by clamping it to the pre-selected
colour
fn color_height(_x: f32, y: f32, _: f32) -> [f32; 4] {
        let mut c = [0.5, 0.5, 0.5, 1.0];

        if y > 80.0 {
                c = [1.0, 1.0, 1.0, 1.0]
        } else if y > 35.0 {
                c = [0.75, 0.75, 0.75, 1.0]
        } else if y > 10.0 {
                c = [0.3921, 0.647, 0.2941, 1.0]
        } else if y > 0.0 {
                c = [0.9764, 0.8941, 0.8196, 1.0]
        } else {
                c = [0.2627, 0.3725, 1.0, 1.0]
        }
```

```
        return c
}

// Generates indecies for a grid of size.0 by size.1
fn gen_idx(size: (u32, u32)) -> Vec<u32> {
   let mut idx: Vec<u32> = vec![0 as u32; (size.0 * size.1 * 6) as usize];

   let mut vert: u32 = 0;
   let mut tris: usize = 0;

   for _ in 0..size.0 {
      for _ in 0..size.1 {
            idx[tris + 0] = vert;
            idx[tris + 1] = vert + 1;
            idx[tris + 2] = vert + size.1 + 1;

            idx[tris + 3] = vert + size.1 + 2;
            idx[tris + 4] = vert + size.1 + 1;
            idx[tris + 5] = vert + 1;

            vert += 1;
            tris += 6;
        }

        vert += 1
    }

    return idx;
}

// Left as an example of flat normals, not actually used anywhere
fn gen_normals(size: (u32, u32), positions: Vec<[f32; 3]>) -> Vec<[f32; 3]> {
        let mut normals: Vec<[f32; 3]> = Vec::new();
        let mut a: [f32; 3];
        let mut b: [f32; 3];


        for i in 0..size.0+1 {
                for j in 0..size.1+1 {
                        a = [0.0, 0.0, 0.0];


                        if j == size.1 {
                                a = vec_sub(positions[(i + j * size.0) as usize],
positions[(i + j * size.0 - 1) as usize]);
```

```
                        }

                        if i == size.0 {
                                b = vec_sub(positions[(i + j * size.0) as usize],
positions[(i + (j + 0) * size.0 + 1) as usize]);
                        } else {
                                a = vec_sub(positions[(i + j * size.0) as usize],
positions[(i + j * size.0 + 1) as usize]);
                                b = vec_sub(positions[(i + j * size.0) as usize],
positions[(i + (j + 1) * size.0 + 1) as usize]);
                        }

                        normals.push(cross_prod(a, b));
                }
        }


        return normals;
}

// Smooth normals generator, averages the perpendiculars to neighboring triangles in
all directions
// if there is an edge, anchor point O is used.
fn gen_smooth_normals(size: (u32, u32), positions: Vec<[f32; 3]>) -> Vec<[f32; 3]> {
        let mut normals: Vec<[f32; 3]> = Vec::new();
        let mut o: [f32; 3];
        let mut a: [f32; 3];
        let mut b: [f32; 3];
        let mut c: [f32; 3];
        let mut d: [f32; 3];


        for i in 0..size.0+1 {
                for j in 0..size.1+1 {

                        o = positions[(i * (size.0 + 1) + j) as usize];


                        // Literal edge cases for each L/R/U/D direction
                        if i == 0 {
                                a = o
                        } else {
                                a = positions[((i - 1) * (size.0 + 1) + j) as usize];
                        }
```

```
                        if j == 0 {
                                d = o
                        } else {
                                d = positions[(i * (size.0 + 1) + j - 1) as usize];
                        }


                        if i == size.0 {
                                c = o
                        } else {
                                c = positions[((i + 1) * (size.0 + 1) + j) as usize];
                        }

                        if j == size.1 {
                                b = o
                        } else {
                                b = positions[(i * (size.0 + 1) + j + 1) as usize];
                        }


                        normals.push(tr_wt_avg(o, a, b, c, d));
                }
        }


        return normals;
}

pub fn inv_idx(idx: Vec<usize>) -> Vec<u32> {
        let mut idx_new: Vec<u32> = vec![0 as u32; idx.len()];
        for i in (0..(idx.len() - 3)).step_by(3) {
                idx_new[i] = idx[i + 2] as u32;
                idx_new[i + 1] = idx[i + 1] as u32;
                idx_new[i + 2] = idx[i] as u32;
        }

        return idx_new
}
```

systems.rs

```
use bevy::prelude::*;
```

```
use crate::chunk::*;

use crate::meshgen;
use crate::ui::*;
use crate::{
        CHUNK_RES,
        SCALE,
        CHUNK_MAX_NUM,
};

// Flush all chunks every 60 secconds
pub fn chunk_timed_flush(
        time: Res<Time>,
        mut query: Query<Entity, With<MeshIndicator>>,
        mut commands: Commands,
        mut chunks: ResMut<ChunksResource>,
) {
        if time.seconds_since_startup() as i32 % 60 == 0 {
                chunks.flush(query, commands)
        }
}

// Scans the surroundings for unloaded chunks and generates them
pub fn chunks_update_sys (
        mut meshes: ResMut<Assets<Mesh>>,
        mut chunks: ResMut<ChunksResource>,
        mut commands: Commands,
        mut materials: ResMut<Assets<StandardMaterial>>,
        mut ui_state: ResMut<UiState>,
        mut camera: Query<&mut Transform, With<Camera>>,
) {
        for cam in camera.iter_mut() {
                // Convert camera position to world integer positon
                let cam_position: (f32, f32) = (cam.translation[0],
cam.translation[2]);

                let mut int_coordinates: (u32, u32) = (0, 0);

                // <0 edge cases
                if cam_position.0 < CHUNK_MAX_NUM.0 as f32 * SCALE {
                        int_coordinates.0 = (cam_position.0 / SCALE) as u32;
                }
                if cam_position.1 < CHUNK_MAX_NUM.1 as f32 * SCALE {
                        int_coordinates.1 = (cam_position.1 / SCALE) as u32
                }
```

```
                if cam_position.0 < 0.0 {int_coordinates.0 = 0};
                if cam_position.1 < 0.0 {int_coordinates.1 = 0};

                // Update UI
                ui_state.int_location = int_coordinates;

                // Check each chunk within the render distance to see if its loaded
                for near_int_coordinates in chunks_load_near(int_coordinates,
ui_state.render_distance) {
                        let idx = (near_int_coordinates.0 + near_int_coordinates.1 *
CHUNK_MAX_NUM.1 as i32) as usize;
                        if chunks.chunks[idx].display == false { // If not loaded
generate it
                                let ChunkMesh = meshgen::gen_ter_mesh(CHUNK_RES,
(near_int_coordinates.0 as f32 * CHUNK_RES.0 as f32, near_int_coordinates.1 as f32 *
CHUNK_RES.1 as f32), ui_state.layers.clone());
                                commands.spawn_bundle(PbrBundle {
                                        mesh: meshes.add(ChunkMesh.clone()),
                                        transform:
Transform::from_scale(Vec3::splat(SCALE / CHUNK_RES.0 as
f32)).with_translation(Vec3{x: near_int_coordinates.0 as f32 * SCALE, y: 0.0, z:
near_int_coordinates.1 as f32 * SCALE}),
                                        material: materials.add(StandardMaterial {
                                                base_color: Color::rgba(1.0, 1.0,
1.0, 0.0),

                                                perceptual_roughness: 1.0,
                                                metallic: 0.0,
                                                ..default()
                                        }),
                                        ..Default::default()
                                }).insert(MeshIndicator{}); // Mesh indicator for
marking to use in despawn later
                                chunks.chunks[idx].display = true;
                                chunks.chunks[idx].mesh = ChunkMesh; // Saved for
export function
                        }

                }

        }
}

// Given (X, Y) generates all chinks within rad radius in manhattan
fn chunks_load_near(pos: (u32, u32), rad: u32)  -> Vec<(i32, i32)> {
```

```
        let mut res: Vec<(i32, i32)> = Vec::new();

        let posi = (pos.0 as i32, pos.1 as i32);

        for i in -(rad as i32)..(rad as i32) {
                for j in -(rad as i32)..(rad as i32) {
                        let mut x = 0;
                        if posi.0 + i > 0 {
                                x = posi.0 + i
                        }
                        if x > CHUNK_MAX_NUM.0 as i32 {
                                x = CHUNK_MAX_NUM.0 as i32
                        }

                        let mut y = 0;
                        if posi.1 + j > 0 {
                                y = posi.1 + j
                        }
                        if y > CHUNK_MAX_NUM.1 as i32 {
                                y = CHUNK_MAX_NUM.1 as i32
                        }

                        res.push((x, y))
                }
        }

        return res
}
```

terrain.rs

```
use noise::{NoiseFn, Perlin};
use crate::{
        PERLIN_FREQ,
        PERLIN_HEIGHT_SCALE
};

// Generates an array of layered perlin noise according to parameters given in
"layers"
pub fn gen_perlin(size: (u32, u32), offset: (f32, f32), layers: Vec<[i32; 3]>) ->
Vec<[f32; 3]> {
        let mut vert: Vec<[f32; 3]> = Vec::new();
        let perlin = Perlin::new();
```

```
        for x in 0..size.0+1 {
                for z in 0..size.1+1 {
                        let mut finally: f64 = 0.0;

                        // Cycle through layers vector that has [frequency,
amplitude, is_enabled]
                        for l in layers.clone() {
                                if l[2] == 1 {
                                        finally += perlin.get([((x as f64 + offset.0
as f64) * PERLIN_FREQ * l[0] as f64), ((z as f64 + offset.1 as f64) * PERLIN_FREQ *
l[0] as f64)]) * PERLIN_HEIGHT_SCALE * l[1] as f64;
                                }
                        }

                        vert.push([x as f32, finally as f32, z as f32]);
                }
        }

        return vert;
}
```

ui.rs

```
use std::string;

use bevy::{prelude::*, render::camera::Projection};
use bevy_egui::{egui, EguiContext, EguiPlugin};
use crate::chunk::*;

// Shared UiState, stores all UI values
#[derive(Default)]
pub struct UiState {
        file_name: String,
        pub render_distance: u32,
        pub int_location: (u32, u32),
        pub layers: Vec<[i32; 3]>,
        l1s: i32,
        l1h: i32,
        l1e: bool,
        l2s: i32,
        l2h: i32,
        l2e: bool,
        l3s: i32,
```

```
        l3h: i32,
        l3e: bool,
        l4s: i32,
        l4h: i32,
        l4e: bool,
        egui_texture_handle: Option<egui::TextureHandle>,
}


// Pre-set defaults for UiState
pub fn ui_state_defaults(
        mut ui_state: ResMut<UiState>
) {
        ui_state.render_distance = 3;

        ui_state.l1s = 7;
        ui_state.l1h = 40;
        ui_state.l1e = true;

        ui_state.l2s = 14;
        ui_state.l2h = 20;
        ui_state.l2e = true;

        ui_state.l3s = 21;
        ui_state.l3h = 10;
        ui_state.l3e = true;

        ui_state.l4s = 28;
        ui_state.l4h = 5;
        ui_state.l4e = true;
}
// TODO: When custom camera controls enable perspective shift
pub fn ui_system(
        mut egui_context: ResMut<EguiContext>,
        mut ui_state: ResMut<UiState>,
        mut query: Query<Entity, With<MeshIndicator>>,
        mut commands: Commands,
        mut chunks: ResMut<ChunksResource>,
) {

egui::SidePanel::left("left_panel").default_width(300.0).resizable(true).show(egui_co
ntext.ctx_mut(), |ui| {
                ui.heading("Terrust - a terrain generator");

                ui.horizontal(|ui| {
```

```
                ui.label("Render distance in chunks");
            });


            ui.horizontal(|ui| {
                ui.add(egui::Slider::new(&mut ui_state.render_distance,
1..=8));
                if ui.button("-").clicked() {
                    ui_state.render_distance += 1;
                }
                if ui.button("+").clicked() {
                    ui_state.render_distance += 1;
                }
            });
            ui.allocate_space(egui::Vec2::new(0.0, 50.0));

            ui.horizontal(|ui| {
                ui.label("Perlin layers (frequency | height | enable)");
            });

            ui.horizontal(|ui| {
                ui.label("#1");
                ui.add(egui::Slider::new(&mut ui_state.l1s, 0..=64));
                ui.add(egui::Slider::new(&mut ui_state.l1h, 0..=64));
                ui.checkbox(&mut ui_state.l1e, "");
            });

            ui.horizontal(|ui| {
                ui.label("#2");
                ui.add(egui::Slider::new(&mut ui_state.l2s, 0..=64));
                ui.add(egui::Slider::new(&mut ui_state.l2h, 0..=64));
                ui.checkbox(&mut ui_state.l2e, "");
            });

            ui.horizontal(|ui| {
                ui.label("#3");
                ui.add(egui::Slider::new(&mut ui_state.l3s, 0..=64));
                ui.add(egui::Slider::new(&mut ui_state.l3h, 0..=64));
                ui.checkbox(&mut ui_state.l3e, "");
            });

            ui.horizontal(|ui| {
                ui.label("#4");
                ui.add(egui::Slider::new(&mut ui_state.l4s, 0..=64));
                ui.add(egui::Slider::new(&mut ui_state.l4h, 0..=64));
```

```
                        ui.checkbox(&mut ui_state.l4e, "");
                });
                ui.allocate_space(egui::Vec2::new(0.0, 50.0));

                // Reloads chunks by flushing all of them, chunk_update_system
immidiatelly regenerates them
                if ui.button("Reload chunks").clicked() {
                        println!("Flush button.");
                        chunks.flush(query, commands);
                        ui_state.layers = vec![
                                [ui_state.l1s, ui_state.l1h, if ui_state.l1e {1} else
{0}],
                                [ui_state.l2s, ui_state.l2h, if ui_state.l2e {1} else
{0}],
                                [ui_state.l3s, ui_state.l3h, if ui_state.l3e {1} else
{0}],
                                [ui_state.l4s, ui_state.l4h, if ui_state.l4e {1} else
{0}]
                        ]
                }

                ui.label(format!("({}, {})", ui_state.int_location.0.to_string(),
ui_state.int_location.1.to_string()));

                ui.allocate_space(egui::Vec2::new(0.0, 50.0));

                ui.horizontal(|ui| {
                        ui.label("File Name: ");
                        ui.text_edit_singleline(&mut ui_state.file_name);
                });

                ui.horizontal(|ui| {
                        if ui.button("Save to OBJ file").clicked() {
                                println!("Save button pressed.");
                                chunks.save(ui_state.int_location,
ui_state.file_name.clone())
                        };
                });

                ui.with_layout(egui::Layout::bottom_up(egui::Align::Center), |ui| {
                        ui.add(egui::Hyperlink::from_label_and_url(
                                "This project is available on github",
                                "https://github.com/idf3da/terrust/",
                        ));
                });
```

```
                ui.allocate_rect(ui.available_rect_before_wrap(),
egui::Sense::hover());
        });
}
```