ENIGMA

A FREE, OPEN SOURCE & EDUCATIONAL GAME DEVELOPMENT KIT

Google Summer of Code

Procedural Fragment Shader Generation Using Classic Machine Learning

Contact Info

Name (Preferred)	Saif Salah Eldeen Kandil (**Saif**)
Date	14 March, 2024
Phone	
Email	
Website	https://k0t0z.github.io/
Discord Username	

Table of Contents

About ENIGMA

ENIGMA is an open-source 2D/3D game development software based on the popular YoYo Game's GMS (GameMaker Studio). ENIGMA uses a Drag & Drop system as well as its programming language. EDL (ENIGMA Development Language) as it is normally called, is a mix between C++ and GML (Gamemaker Machine Language). EDL offers many powerful features and functions.

Abstract

Have you ever wondered how games such as No Man’s Sky created their game objects? Well, Almost all objects in the No Man's Sky universe are procedurally generated, see Figure 1. The term procedural refers to the process when shaders generate textures or effects algorithmically rather than using pre-made images. Fractals are geometric patterns that can often be generated procedurally. Commonplace procedural content includes textures and meshes. Sound is often also procedurally generated. No Man’s Sky might not be everyone’s cup of tea but its code is pretty impressive. Minecraft is another example. Minecraft uses Procedural Generation to algorithmically generate terrain, biomes, and features and decides which blocks are placed where. See Figure 2.

Noise kernels play a vital role in this project such as Perlin. When it comes to creativity in using noises, I always use this cool Rainforest shader by Inigo Quilez which is generated procedurally. Quilez created a beautiful terrain only by using mathematics and pseudorandom noise functions.

After this project, ENIGMA’s users will be able to use Machine Learning features to generate perfect shaders that can be converted into cool textures, see Figure 3. To generate shaders using Machine Learning, we need to implement a way to force the model to generate a valid code, otherwise, it will be completely unreliable because AI models never stop making mistakes.

I will target fragment shaders in this project only, however, I have decided on a design that can be extended to support other types of shaders such as vertex shaders.

Machine Learning will be used to estimate the most effective seed for the generated shaders using GP (Genetic Programming). For example, the same content exists at the same places for all players in No Man’s Sky (thanks to a single random seed number in their deterministic engine), which enables players to meet and share discoveries [4].

This project will only focus on shaders that can be converted into 2D textures easily, such as in Figure 3, as there are many ways that texture data is used to render objects.

Figure 1: No Man’s Sky Procedural Generated Terrain

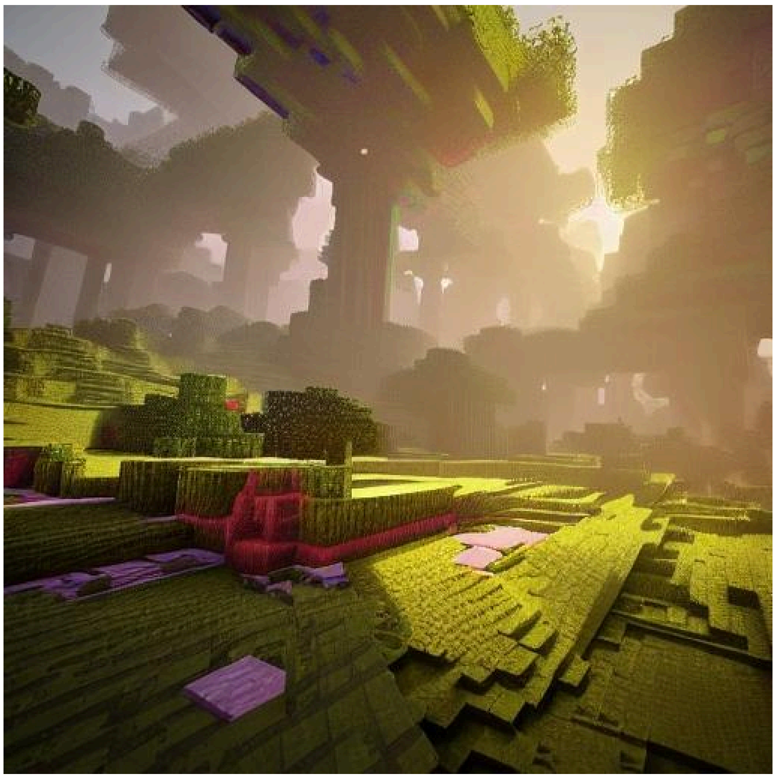
Figure 2: Minecraft Procedural Generated Terrain

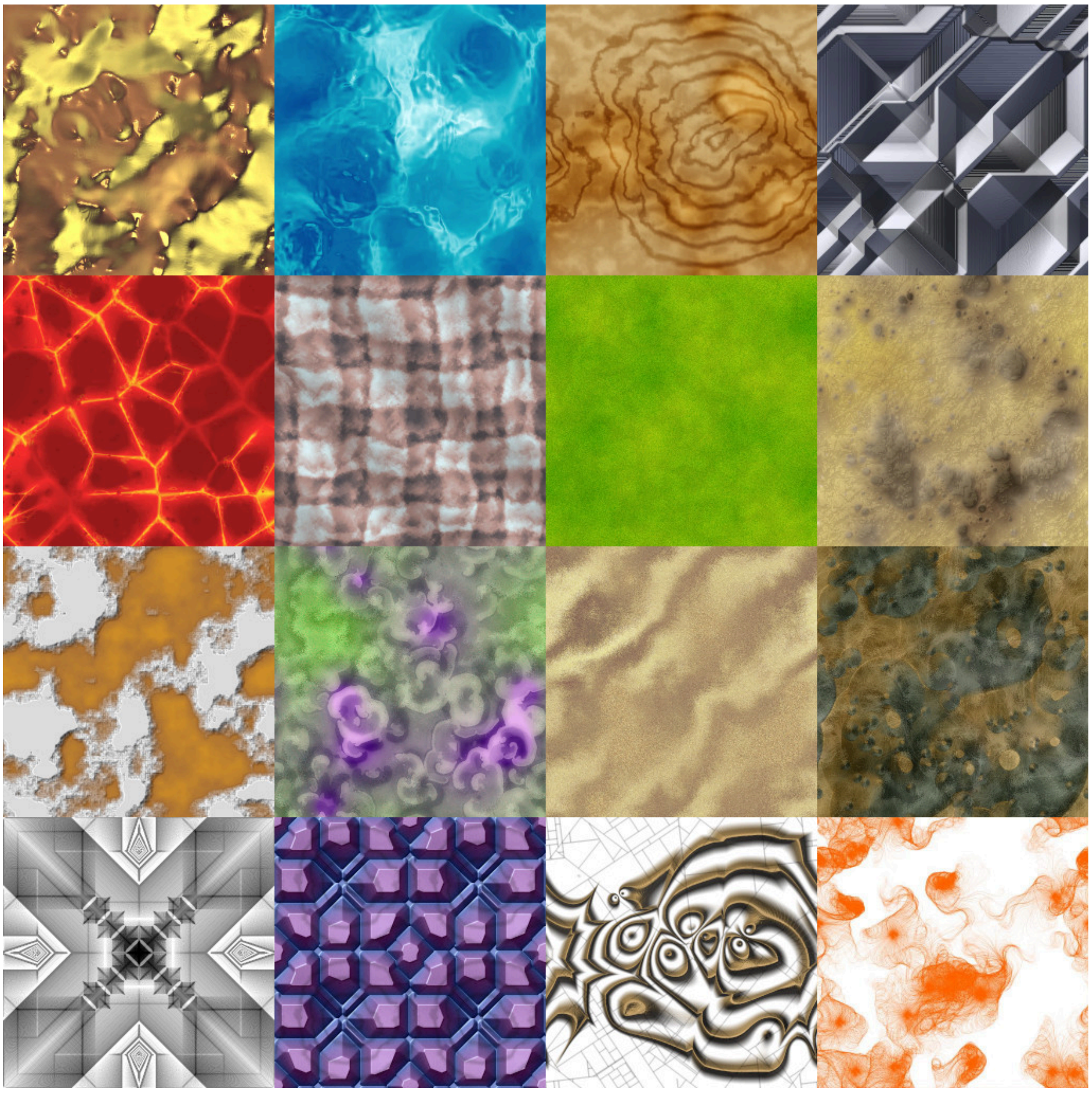
Figure 3: Examples Of Procedural Generated Textures [4]

Architecture

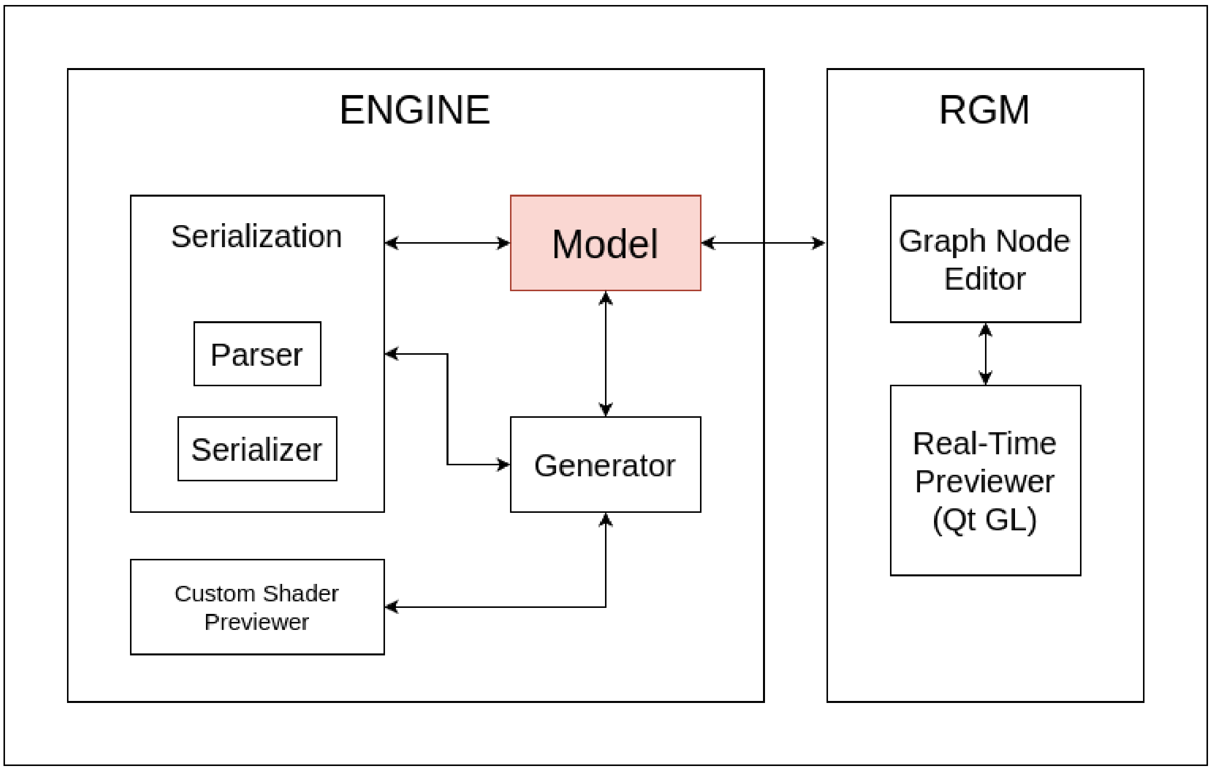

Figure 4: Project Architecture

Model	This will hold the graph representation of nodes and their relationships. In addition, it will provide an easy interface for adding, modifying, and removing nodes from the graph.
Serialization	This will serialize and parse the graph to and from JSON.
Generator	This will generate the shader code from a given graph.

My Approach

After closely examining ENIGMA’s Graphics System, I organized this project into 4 phases. I will go through each one starting from the first.

Phase 1: Improving ENIGMA’s Graphics System

Currently, ENIGMA does support shaders however, it does not have the tooling for extending. In this phase, I will try to flush ENIGMA’s Graphics System and implement the shader generation required modules. This will be the largest phase and may last till after Midterm Evaluations.

Implementation

1. Flushing ENIGMA’s Graphics System.
2. The main model of my architecture will be implemented. It will provide an interface for dealing with the graph as follows:
    - add_node();
    - modify_node(id);
    - delete_node(id);

    Many node types will be implemented including:
    - FloatOp (Add, Sub, Divide, Multiply, …etc)
    - FloatFunc (Round, Floor, Ceil, Sin, Cos, …etc)
    - Texture2D
    - Input
    - Output

3. Implementing the serialization module. The output JSON is something similar to this:

```
{
 "nodes": [
   {
     "id": "floatop_0",
     "type": "floatop",
     "parameters": {
       "operation": "multiply"
     },
     "inputs": [
       { "id": "floatop_0_in_0", "is_connected": true, "value": null },
       { "id": "floatop_0_in_1", "is_connected": true, "value": null }
     ],
     "output": "floatop_0_out"
```

```
    },
    {
      "id": "floatfunc_0",
      "type": "floatfunc",
      "parameters": {
        "function": "round"
      },
      "inputs": [{ "id": "floatfunc_0_in_0", "is_connected": true, "value": null }],
      "output": "floatfunc_0_out"
    },
    {
      "id": "texture2d_0",
      "type": "texture2d",
      "parameters": {
        "width": 512,
        "height": 512
      },
      "inputs": [{ "id": "texture2d_0_uv", "is_connected": true, "value": null }],
      "output": "texture2d_0_out"
    }
  ]
}
```

4. The generator module will be implemented. According to my estimation, this step will be the largest. In addition, this step will not be closed in this phase, however, I will close it inside phase 2. In this step, I will write the shader code that will be generated for each node. For example, the below code is to generate a variable assignment:

```
std::string generate_code(unsigned mode, unsigned type, int id, const std::string *output_var_name) const {
   return "    " + output_var_name + " = " + itos(constant) + ";\n";
}
```

The indentation first then the variable name followed by the assignment operator and finally the constant integer.

As mentioned in [1], prebuilt functions will be defined for each node representation. Each function will generate a shader code based on its type. For each node, all the inputs are mapped to variables in the shader code and fed into prebuilt functions defined for each node representation. The same process is applied to outputs which are also mapped to internal variables.

Figure 5 illustrates my point. I used the VisualShader in Godot game engine to create a simple effect using a noise function.

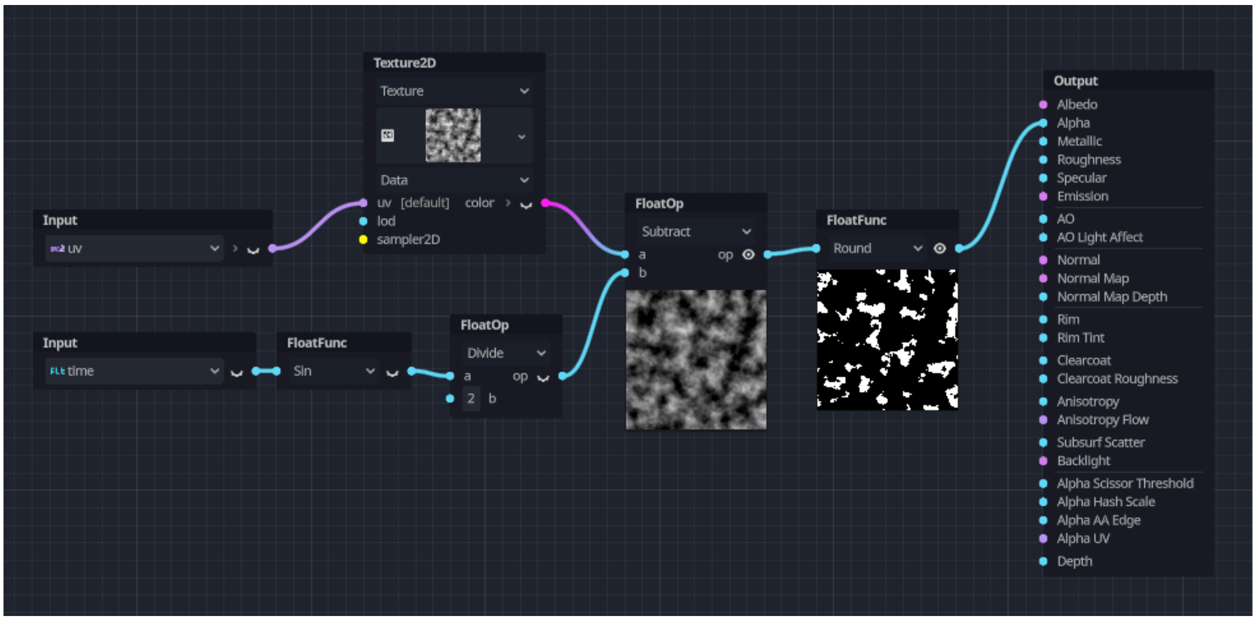

Figure 5: Godot VisualShader

The generated shader for Figure 5 will look like this:

```
shader_type spatial;
render_mode blend_mix, depth_draw_opaque, cull_back,
diffuse_lambert, specular_schlick_ggx;

uniform sampler2D tex_frg_6;

void fragment() {
// Input:2
   vec2 n_out2p0 = UV;

// Texture2D:6
   vec4 n_out6p0 = texture(tex_frg_6, n_out2p0);

// Input:5
   float n_out5p0 = TIME;

// FloatFunc:4
   float n_out4p0 = sin(n_out5p0);
// FloatOp:7
   float n_in7p1 = 2.00000;
   float n_out7p0 = n_out4p0 / n_in7p1;

// FloatOp:8
   float n_out8p0 = n_out6p0.x - n_out7p0;

// FloatFunc:9
   float n_out9p0 = round(n_out8p0);

// Output:0
   ALPHA = n_out9p0;
}
```

This shader code tunes the alpha value of the mesh. This means if the mesh is a sphere, you will have a sphere that can disappear nicely.

5. Implementing a Sampler that grabs the pixel data and colors the point. A way for testing the input shader must be implemented. We need to see the output of each shader without a GUI. A possible solution to this is to use a framebuffer object (AKA Render Target or Surface) to rasterize it. This is the Custom Shader Previewer in the project architecture.

The tricky part is in the rendering part itself. After the shader code is written, a rendering server (As Godot community named it) will be implemented to draw that shader. I will do more investigation into this before May 26.

ENIGMA has its own rendering server which I will try to extend.

Phase 2: Implementing Noise Kernels

This phase can be part of Phase 1 but I preferred writing it as a separate phase because it marks the official start of the project. This is because this project aims to apply Machine Learning to the graph tree structure to find the best seeds for noise functions along the whole tree. The following steps are required in this phase:

1. Implementing the noise kernels from scratch as requested by ENIGMA's admins. This will have some benefits I will mention later. We can integrate existing libraries such as FastNoiseLite, AccidentalNoise, and LibNoise.
2. [Optional] Implementing the noise module. This will facilitate the communication between the newly added noise extension and the engine. I haven't decided yet where this module will go into the codebase. This will be decided before May 26.
3. Extend the generator module with the newly added noise functions. This will close the generator module and open the testing phase.

Infinite Texture Generation With Voronoi (Worley) Noise Challenge

This is another reason for implementing noise kernels from scratch. Voronoi noise can not generate infinite textures without using Tilling. If we have N points created randomly, I either need to tile those points or I will not be able to generate pixels beyond their containing box so the texture is finite.

As advised by my mentor, we can expand the tileability by introducing more octaves. This means 50% of points can't leave their grid cell, 75% of points can't leave the surrounding 8x8 cells, 87.5% of points can't leave the 64x64 surrounding cells, and 93.75% can't leave the 512x512 region, and so on.

I can create a more complex and visually interesting texture by layering multiple octaves with varying frequencies and amplitudes. Additionally, this layering can help mitigate the issue of finite boundaries.

Implementation

As I said earlier, this step is part of phase 1 in which we will have separate nodes for each kind of noise function. For example, this is a function that generates a shader code capable of generating a smooth value noise:

```
std::string generate_value_noise() {
   std::string value_noise = R"(
       float noise_random_value(vec2 uv) {
           return fract(sin(dot(uv, vec2(12.9898,
78.233)))*43758.5453);
       }

       float noise_interpolate(float a, float b, float t) {
           return (1.0-t)*a + (t*b);
```

```
        }

        float value_noise(vec2 uv) {
            vec2 i = floor(uv);
            vec2 f = fract(uv);
            f = f * f * (3.0 - 2.0 * f);

            uv = abs(fract(uv) - 0.5);
            vec2 c0 = i + vec2(0.0, 0.0);
            vec2 c1 = i + vec2(1.0, 0.0);
            vec2 c2 = i + vec2(0.0, 1.0);
            vec2 c3 = i + vec2(1.0, 1.0);
            float r0 = noise_random_value(c0);
            float r1 = noise_random_value(c1);
            float r2 = noise_random_value(c2);
            float r3 = noise_random_value(c3);

            float bottomOfGrid = noise_interpolate(r0, r1, f.x);
            float topOfGrid = noise_interpolate(r2, r3, f.x);
            float t = noise_interpolate(bottomOfGrid, topOfGrid, f.y);
            return t;
        }

        void simple_noise_float(vec2 uv, float scale, out float
out_buffer) {
            float t = 0.0;

            float freq = pow(2.0, float(0));
            float amp = pow(0.5, float(3-0));
            t += value_noise(vec2(uv.x*scale/freq,
uv.y*scale/freq))*amp;

            freq = pow(2.0, float(1));
            amp = pow(0.5, float(3-1));
            t += value_noise(vec2(uv.x*scale/freq,
uv.y*scale/freq))*amp;

            freq = pow(2.0, float(2));
            amp = pow(0.5, float(3-2));
            t += value_noise(vec2(uv.x*scale/freq,
uv.y*scale/freq))*amp;
```

```
            out_buffer = t;
        }
    )";
    return value_noise;
}
```

If you want to play around with it, I created a Shadertoy version of it at k_value_noise.

Phase 3: Implementing a Graph Node Editor for RGM

Here in this phase, depending on the discussions that will be made before May 1, I will implement a visual graph editor from scratch or integrate an existing one into RGM.

Implementation

This phase will have these expected detailed steps:

1. Implementing the graph editor in RGM using Qt5. We can integrate existing ones such as paceholder/nodeeditor. It is designed to be a general-purpose graph visualization and modification tool, without specialization on only data propagation and it is a perfect fit for this project. The integration is much easier as this node editor is CMake-based and compatible with Qt5 which RGM depends on.
2. Implementing the integration between the model and the visual editor.
3. Testing the full system (Integration Testing).

Phase 4: Generating Shaders Using Genetic Programming

Although it is fine if I did not make it to this point, this phase is considered the core of the project and it is vital to ENIGMA because it will introduce it to the world of Machine Learning. This will be another reason for me to extend my period. See the timeline and my commitments below.

In this phase, we will generate an approximate shader for a specific picture. For example, if I want to generate an ice texture from a shader that we can use for mountains, I will put one as an input to the model.

A dataset of images containing the required art we want such as walls, floors, ceils, and others that we can use for effects such as fire and the ground of a swimming pool will be collected and used for training the model using a Genetic Programming approach.

A Clustering algorithm will seed initial colors and noise kernels, and then a genetic algorithm will take over to find the best actual matches.

The cost function will be the difference between the output texture and the input texture. According to my research, $MSE(y, \widehat{y})$ will do the job as this is not a classification task.

Implementation

1. Define the important aspects of the Genetic algorithm including:
    - The Genetic representation of seeded noise kernels and other information that is required to be encoded.
    - The function that will generate new solutions for new generations.
    - The Fitness (Cost) function. It will be used to evaluate new solutions.
    - The Selection function will select new solutions.
    - The Crossover function.
    - The Mutation function.

2. Define success criteria.
3. Prepare our datasets.
4. Train the model.
5. Deploy the trained model.

Note that I mark this phase as future work as it is already huge enough. In addition, I intend to publish a paper, if we have achieved outstanding results.

I have to mention that **The ENIGMA Team** organization will buy a Cloud share for deploying the trained models in case of achieving noticeable results.

Note that the final dataset specifications are not accurately determined.

Stretch Goals

1. A text editor besides our graph editor will be a great choice to give game developers great flexibility while writing shaders. This step is crucial as the graph node editor will not expose all features of the shader script so using both in parallel might be necessary for specific effects [3].
2. Adding support for a custom node creation for the visual editor graph.

Deliverables

1. A fully functional visual graph shader editor with 2D texture generation capability (**Required**).
2. A Machine Learning-based algorithm for finding the best seed for our shaders (**Stretch Required**).
3. A visual shader text editor to give game developers more flexibility while writing shaders (**Optional**).

Timeline

Before May 1

- Decide whether to implement a graph editor from scratch or integrate an existing one. A great choice might be paceholder/nodeeditor.
- Write some QTests for RGM in RadialGM/tests/.
- Investigate the possibility of adding custom widgets inside paceholder/nodeeditor as we need to preview shader output at every node from a Qt GL context.
- Investigate ENIGMA's Graphics System again more carefully.
- Work with admins on a good approach for my previous Google Summer of Code 2023 project as in #2388.
- Maybe work on moving my Google Summer of Code 2023 project's documentation to the official ENIGMA wiki.
- Try to merge #2358 as the serialization module will depend on ENIGMA's JSON extension.
- Write the project idea as this is not written on the website.
- Getting schoolwork done so I have less work after May 1.

May 2 - 26

- Read more about the architectures of a shader renderer and previewer.
- Decide with mentors whether we are going to use the maintained buffer work as the target branch will depend on it. See #2361.
- Investigate if using an existing noise library is better than implementing one from scratch.
- Investigate if using an existing general-purpose visual graph editor is better than implementing one from scratch.
- Set up my work environment. Maybe start implementing the main model architecture.
- Discuss the limitations of the current Graphics System.
- This is a great period for getting schoolwork done. Having a lot of schoolwork in parallel with both final exams and the Google Summer of Code project after May 26 could be better (check my commitments).

May 27 - June 5
(Coding officially begins!)

- Implement the main architecture.
- Design a proper testing procedure to test the main module.
- Write documentation.
- Review the deployment plan.

June 6 - 15

- Implement the serialization module.
- Design a proper testing procedure to test the serialization module.
- Write documentation.

June 16 - 30

- Implement the generator.
- Extend the testing procedure to test the generator.
- Write documentation.
- Close phase 1.
- Implement the noise functions.
- Extend the generator with the noise functions.
- Close phase 2.

July 1 - 7

- A saved period for any unexpected latency.
- Improve documentation and unit tests.
- Improve the testing strategy
- Review the final product output with mentors.

July 8 - 12 (Midterm Evaluations)

(Standard Coding Period)

- Discuss a good approach for submitting the work done with mentors. This is to prevent 10k+ lines from being added to any of my PRs.
- Submit PRs for review.
- Finalize midterm blogs.
- Populate the ENIGMA wiki with the written documentation.
- Discuss the testing strategy with mentors.

July 13 - 20

- Check paceholder/nodeeditor for integration.

July 21 - August 18

- Integrate paceholder/nodeeditor into RGM.
- Implement the custom widgets for my use case as paceholder/nodeeditor is general-purpose.
- Implement the binding strategy using gRPC.
- Design an integration test plan to test the full system.
- Document the strengths and weaknesses of the system.
- Finalize the testing phases.
- Write documentation.
- Close phase 3.
- Discuss the possibility of working on phase 4.
- [Optional] Review [1] again more carefully.

August 19 - 26 (Final Evaluations)

(Standard Coding Period)

- Polish final PRs for ENIGMA and RGM.
- Write the final report and the **Wrapping Up** post in my blog.
- Finalize my blog posts for the summer of 2024.
- Close the documentation phases.

Note that this timeline is estimated using **Standard Coding Period** dates. I may extend this project which means all of the periods mentioned in the timeline will be increased.

Contribution to Open Source

Additional Commitments

Commitment	From	To	Max. Hours/week
2nd Semester Final Exams (Graduation Year)	25 May, 2024	14 June, 2024	12
Military Service			

Note: In case I register for a Master's degree (which I intend to do), the Military Service commitment will not be applicable anymore, and I will be able to extend my project to 22 weeks which is also what I intend to do.

Future Work

I have to mention that phase 4 in this proposal is a project on its own but I hope we can get some of its work done. However, I will mark it as a future work for ENIGMA as Deep Learning can be applied with Neural Networks and LLMs for estimating precise parameters to generate a perfect shader code.

Another future work is to support Vertex shaders as well as other shader types to give ENIGMA's users more control over the generated shaders.

I will be happy to mentor this project and its resulting projects in the future with **The ENIGMA Team** organization as I mainly work as a Machine Learning Engineer.

References

[1] Sasso, Elio, Daniele Loiacono, and Pier Luca Lanzi. "A Tool for the Procedural Generation of Shaders using Interactive Evolutionary Algorithms." arXiv preprint arXiv:2312.17587 (2023).

[2] Helsing, Johan K., and Anne C. Elster. "Noise modeler: An interactive editor and library for procedural terrains via continuous generation and compilation of GPU shaders." In Entertainment Computing-ICEC 2015: 14th International Conference, ICEC 2015, Trondheim, Norway, September 29-Ocotober 2, 2015, Proceedings 14, pp. 469-474. Springer International Publishing, 2015.

[3] Godot Engine Documentation.

[4] Wikipedia contributors, "Procedural generation," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Procedural_generation&oldid=1215018326.

Procedural Fragment Shader Generation Using Classic Machine Learning

As mentioned in [A Tool for the Procedural Generation of Shaders using Interactive Evolutionary Algorithms
](https://arxiv.org/abs/2312.17587) paper, this project is divided into two major steps:

1. Implement the visual shader graph functionalities.
2. Apply a proper Genetic Algorithm to find the best shader for a pre-specified image.

The contributor may also read [Noise Modeler: An Interactive Editor and Library for Procedural Terrains via Continuous Generation and Compilation of GPU Shaders](https://link.springer.com/chapter/10.1007/978-3-319-24589-8_42) paper for a better understanding of the graph editor architecture.

ENIGMA's graphics system