VISUAL SHADER PROGRAMMING

Metelitsa D. S., Savenko A. G.
Institute of Information Technologies of the Belarusian State University of Informatics and Radioelectronics
Minsk, Republic of Belarus
E-mail: savenko@bsuir.by

The paper presents a developed software tool for visual programming of shaders, designed to automate the process of visual description of shaders by technical artists and programmers in order to speed up the prototyping process. The result of the development will be visually presented immediately. The use of this tool by technical artists will reduce the interaction between the artist and the programmer in the process of working on a project, since the software tool does not require programming knowledge or other specific skills from the user. The result of shader programming will be export to GLSL-code, compatible with a large number of game engines.

Introduction

The development of technology leads not only to the simplification of the work of a modern person, but also to an increase in the quality of the entertainment sector. Computer games and graphics in cinematography have long been attractive not only for children, but also for a much wider audience. The consumer makes more and more demands on the quality of digital content, the level of image modeling, the quality of detailing of graphic images. The modern IT-sphere has long gone beyond the solution of purely mathematical problems.

I. Computer graphics processing

A shader is a program that runs on a graphics card's graphics processing unit (GPU). The shader receives input data containing information about vertex coordinates, polygons, normals, lighting, vertex color, UV (texture coordinates), etc. The task of the shader is to accept this data, process it, and output the final result [1]. One of the basic concepts of shader programming is the 3D model. It includes two main items: vertex and texture. Each vertex has its own coordinates, as well as a normal. The vertices are combined into polygons using edges, and the polygons, in turn, form a polygonal mesh. A texture is a simple image that is positioned on the model according to UV coordinates. UV coordinates - the correspondence between the coordinates on the surface of a three-dimensional object (X, Y, Z) and the coordinates on the texture (U, V) (Figure 1). The values of U and V usually range from 0 to 1. That is, each vertex of the model has its corresponding coordinates on the texture. The shader is executed for each individual vertex/pixel separately. At the same time, it has information only about the vertex / pixel that it is currently processing. Those. at the time of writing the shader, we don't know what color the adjacent vertex/pixel is. There are pixel and vertex shaders. A vertex shader is a shader that processes vertex data and then passes it to the pixel shader. What data the shader will transmit can be set by the programmer. The vertex shader is executed before the pixel shader, and then data is passed from it to the pixel shader. A pixel shader is a shader that takes interpolated data from a vertex shader and, based on it, calculates the color for each individual pixel [2].

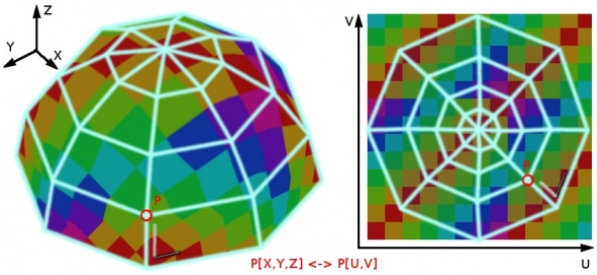

Figure 1 – Correspondence of coordinates (x, y, z) and (u, v)

The rasterizer divides the triangle into pixels, for which the texture coordinates and color are interpolated. Then, for each fragment, the following operations are performed: pixel ownership check, texture mapping, applying fog effects, alpha test, stencil-test, depth test, blending, dithering and boolean operations. After processing all these methods, the resulting fragment is placed in the frame buffer, which is subsequently displayed on the screen. [3]

II. Software Development

The developed software tool for visual shader programming allows to reduce the «artist-programmer» interaction in the process of working on a project, since the software tool does not require programming knowledge or other specific skills from the user. The development of the software was carried out in the C++ programming language. During development, the following modules were implemented:

- UI - is responsible for rendering, interaction with the user interface and the graph built during the creation of the shader.
- Generator - generates GLSL-code according to the graph obtained during the use of the program.
- Render - Renders a circle/square using a base or generated shader.

– Serialization - saves the description of the graph to a file, reads the description of the graph from the file and creates the necessary objects based on this description.
– Utils is a set of utility methods and structures.

The scheme of interaction between software modules is shown in Figure 2.

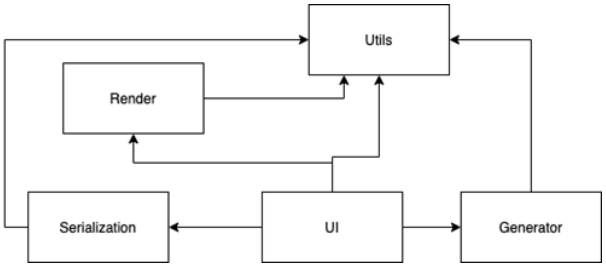

Figure 2 – Scheme of interaction between modules

The development of shaders is based on mathematical algorithms for processing graphs. A shader graph is a node-based GUI that allows designers and artists to add and connect nodes to create a shader without having to write any code. The basic idea is this: a shader graph is a graphical representation of a fragment shader (GLSL in this case). Each node in this graph is a text block in GLSL code. For example, the Multiply node takes two floating point numbers or vectors and returns the result of their multiplication. In this case, a plain text format is used to specify the nodes, instead of describing each node with a class. This approach has a number of advantages: nodes can be implemented quickly, and nodes can be changed or added without recompiling the engine. The nodes and links of the graph built during the creation of the shader are data. Structures are used to describe them. Figure 3 shows the pseudocode of the developed shader node bypass algorithm.

```
function resolve_node(node):
let input_code = ""
let this_code = node.code
store_outputs(node) // create variables for each output
for i in node.inputs:
if i.connected_node is resolved:
replace(this_code, i, i.connected_output)
else:
input_code += resolve_node(i.connected_node)
replace(this_code, i, i.connected_output)
return input_code + this_code
let output = resolve_node(master_node)
```

Figure 3 – Pseudocode of the shader node traversal algorithm

To generate the code, an “input” point is needed, starting from which the graph will be bypassed. The node with which it is connected is determined along the link, and for that node, also find the one with which it is connected (i.e., we bypass the adjacent vertices of the graph sequentially). Thus, initially we reach the node without connections at the input, we write its code into a string, and so on until we return back to the color output node, and we actually get the code described by the graph. The interface of the software tool is similar to well-known and common shader editors. The workspace is a grid, on the left is the preview area, at the top is the main menu. Adding nodes of the shader graph is carried out using the context menu (right-clicking and selecting the node type) (Figure 4).

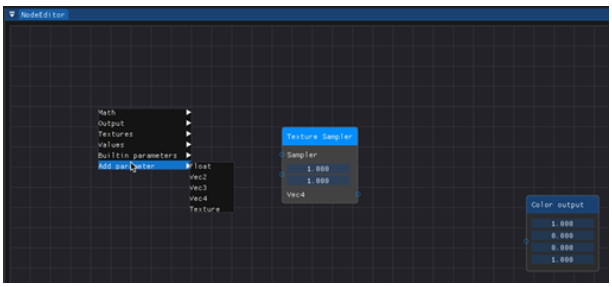

Figure 4 – Adding Nodes

When adding textures to a graph, a dialog box is used that allows you to load textures from files. Each pixel of a geometric object is painted with the corresponding texture color. The output is a three-dimensional texture effect that is correctly perceived by the viewer. Figure 5 shows how the texture is applied to the geometry of the object (in this case, in the preview mode, the user can use the mouse to change the angle of rotation of the object to view all the applied effects).

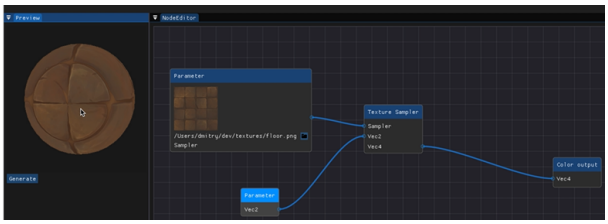

Figure 5 – Applying a Texture to an Object

One of the most important functions of the software is to export the developed shader to GLSL code. It is this feature that allows you to combine shaders created in the program with various custom engines. Separately, it is worth mentioning that the use of this PS does not require the presence of the Internet and the installation of any additional software. As a result, a software tool for visual programming of shaders was developed, which practically does not require programming knowledge from the user and has a convenient minimalistic interface and a discreet, business-like design style.

III. References

1. Dev Tribe. Fundamentals of shader programming [Electronic resource]. - Access mode: https://devtribe.ru/p/unity/quick-theory-of-shaders-1. Access date 09/20/2022.
2. Habr.com. Programming shaders in Unity [Electronic resource]. - Access mode: https://habr.com/ru/post/474812. Access date 09/20/2022.
3. GameDev.ru Shader programming in HLSL [Electronic resource]. - Access mode: https://gamedev.ru/code/articles/HLSL. Access date 09/20/2022.