A Survey of GLSL Examples

Thiago Gomes, Luiz Estevão, Rodrigo de Toledo
Programa de Pós Graduação em Informática, IM
Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brasil
{telias,lfestevao,rtoledo}@dcc.ufrj.br

Paulo Roma Cavalcanti
COPPE Sistemas and
Department of Computer Science
Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brasil
promac@gmail.com

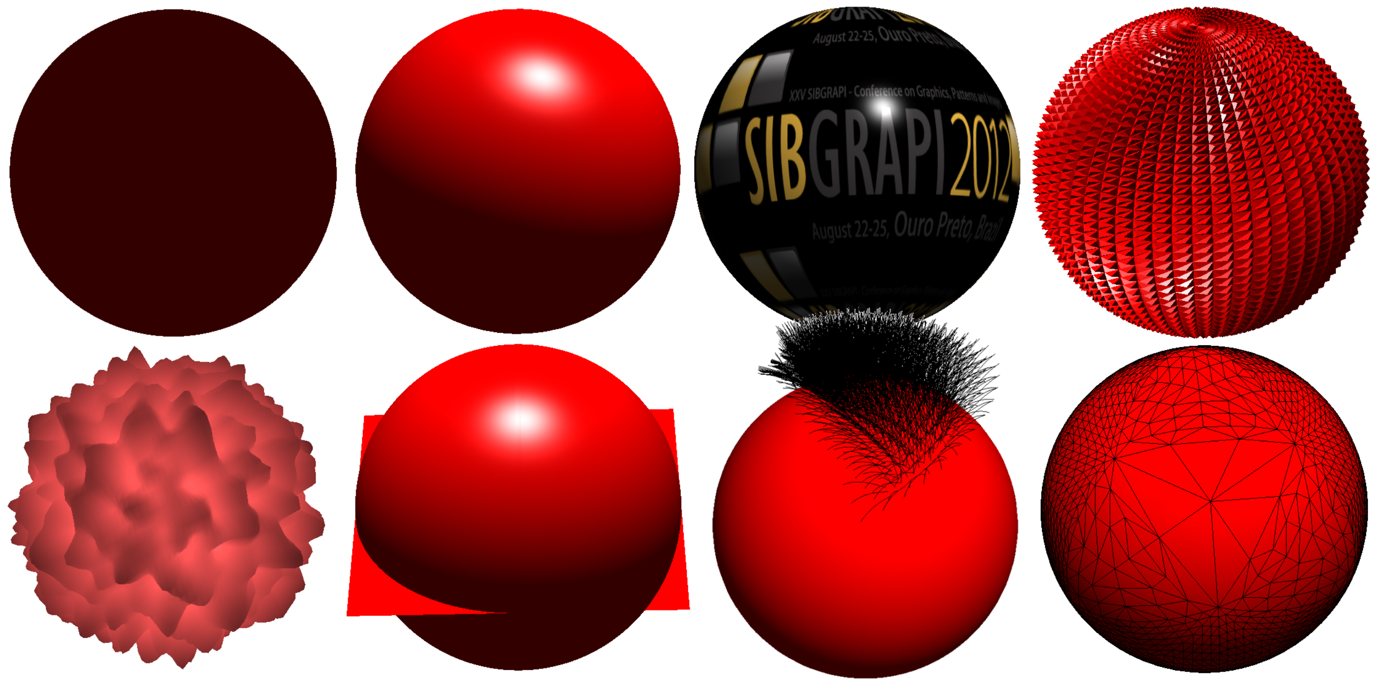

August 22-25, Ouro Preto
XXV SIBGRAPI - Conference on Graphics, Patterns and Images
SIBGRAPI 2012
August 22-25, Ouro Preto, Brazil

Fig. 1. Results from different shader examples. In the first line, there are some basic examples (from left to right): vertex positioning; Phong model; texture application; and triangle extrusion using a geometry shader. In the second line, there are some improved examples (from left to right): vertex noise applied on a shpere using a vertex shader; sphere ray casting from a box using a fragment shader; a "mohican" hair using a geometry shader; and continuous LOD sphere using tessellator shaders.

***Abstract*—This survey provides GLSL information for beginners, by means of a series of commented codes and technical explanations, providing an effective way for learning GLSL, one of the main multi-platform and multi-hardware shader programming languages available. The examples increase in complexity through the text. They may run in several shader tools, including Shaderlabs, a shader development environment, which was developed aiming at helping those willing to learn and practice GLSL development.**

***Keywords*- GLSL; ShaderLabs; Shaders; Graphics Pipeline**

I. Introduction

State of the art graphic applications often use some of the modern video cards' programmable stages. As a consequence, the comprehension of the most advanced techniques require some knowledge about the graphics pipeline.

The focus of this survey is on GLSL programming. We strongly suggest the use of an IDE for GPU programming (see Section II-A) to try the examples described in the following sections. For this reason, we do not show any OpenGL/C++ code or approach, but a series of GLSL examples. Each time, the code is presented before its explanation, allowing the reader to learn by example. In the case of a class or group, we suggest the use of Coding Dojo [1], which provides an effective learning environment. The following audience is addressed:

- **CG starters:** by teaching in a practical, but simple way, concepts like the Fixed-Function Graphics Pipeline, teh Programmable Graphics Pipeline, and the GLSL syntax.
- **Professionals:** by presenting a set of the most popular Shader techniques.
- **Academics:** by bringing a collection of the most recent papers and techniques.
- **Teachers:** by showing a successful path applied to Shader learning.

A. *Survey overview*

Section II lists some references about GLSL and also some shader programming tools. After describing the graphics card evolution in Section III, Section IV presents some simple examples, aiming at the teaching of Shader languages, thus helping the understanding of the Graphics Pipeline, and the GLSL syntax. It is recommended the use of a tool for abstracting the setup code, necessary in this level, and we have developed ShaderLabs [2] for this purpose. Being a free, open software, and currently the only tool that supports coding and testing in all stages, it is the ideal tool for experimenting all sort of shader techniques. Section V presents some useful techniques, explaining more in depth each programmable stage. Finally, Section VI concludes the survey.

II. Related Work

The programmable pipeline is essential to produce real-time visual effects and to accelerate graphics algorithms. However, there are few valuable sources on how to program in GPU, especially the most recent shaders. Rost [3] presents details about vertex and fragment shaders and how to setup them in an OpenGL application. Marroquim and Maximo tutorial [4] has a good introduction to GLSL, including structured figures about the pipeline and simple examples. However, there is no information about the more recent tessellation shaders. In NeHe site [5] there is a practical introduction to GLSL, and the tutorials found in LightHouse3D site [6] contain significant information, but all without the tessellation shaders. Specific information about the tessellation shaders can be found in a few sites [7] and in Valderato et al. tutorial [8] for DirectX.

A. *GPU programming tools*

The usual steps in a graphics application that uses GLSL shaders are:

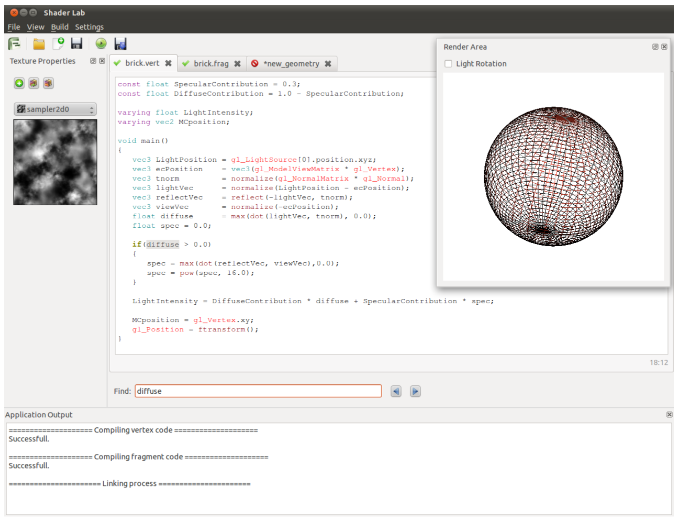
1) Code the application in a common language such as C++;
2) Use a graphics API;
3) Load a 3D model, and send it to the graphics card;
4) Set the parameters necessary to a 3D scene: camera position, illumination, object position and rotation.
5) Code the shaders and send them to the graphics card;

In this survey, we focus on how coding the shaders, which is only a part of the item 5 above.

There are some IDE (Integrated Development Environment) specific for GPU programming. With an IDE, a programmer may only focus on writing shader codes. This is a very helpful tool for those who are learning, testing or evaluating GPU algorithms. Most tools use the GLSL shader language and support the vertex and fragment programmable stages. Some examples are: Shader Designer[9], Render Monkey[10], Shader Maker[11] and ShaderLabs[2]. These tools are compared in Table I

	Shader Maker	Render Monkey	Shader Designer	ShaderLabs
Vertex/Fragment	X	X	X	X
Geometry	X			X
Tessellation				X
Multiple textures	X	X	X	X
Multiple objects		X		X
Shader language	GLSL	GLSL/HLSL	GLSL	GLSL
Preset effects		X		

TABLE I
Comparative table about the GPU programming tools. It should be noted that only ShaderLabs supports tessellation shaders.



```
const float SpecularContribution = 0.3;
const float DiffuseContribution = 1.0 - SpecularContribution;

varying float LightIntensity;
varying vec2 MCposition;

void main()
{
   vec3 LightPosition = gl_LightSource[0].position.xyz;
   vec3 ecPosition    = vec3(gl_ModelViewMatrix * gl_Vertex);
   vec3 tnorm         = normalize(gl_NormalMatrix * gl_Normal);
   vec3 lightVec      = normalize(LightPosition - ecPosition);
   vec3 reflectVec    = reflect(-lightVec, tnorm);
   vec3 viewVec       = normalize(-ecPosition);
   float diffuse      = max(dot(lightVec, tnorm), 0.0);
   float spec = 0.0;

   if(diffuse > 0.0)
   {
      spec = max(dot(reflectVec, viewVec),0.0);
      spec = pow(spec, 16.0);
   }

   LightIntensity = DiffuseContribution * diffuse + SpecularContribution * spec;

   MCposition = gl_Vertex.xy;
   gl_Position = ftransform();
}
```

```
==================== Compiling vertex code ====================
Successfull.

==================== Compiling fragment code ====================
Successfull.

====================== Linking process ======================
```

Fig. 2. ShaderLabs IDE.

III. Graphics Pipeline Evolution

A. *Graphics Pipeline*

A pipeline is a hardware technique used to speed up programs that repeatedly applies expensive operations, which can be divided into several independent and sequential steps. Shaders are the programmable stages of the graphic pipeline.

In 3D applications, many similar operations are performed onto different vertices, and thus a graphic pipeline can be used for optimization. The graphic pipeline can be executed by a 3D video card, an exclusive hardware, which boosts the performance, but lacks in flexibility. The fixed-function graphics pipeline (not programmable) is described in the sequel.

In order to transfer the geometry from the application to the video card, vertices are sent through a transformation pipeline. Each stage uses one type of coordinate system or changes the coordinate system into another one. Fig. 3 presents the pipeline [12], but distinct manufacturers can implement the pipeline differently.

Usually, models have vertices defined in its own coordinate system, centred at the origin. The first stage of the pipeline, as shown in Fig. 3, sets global coordinates to the model's vertices,

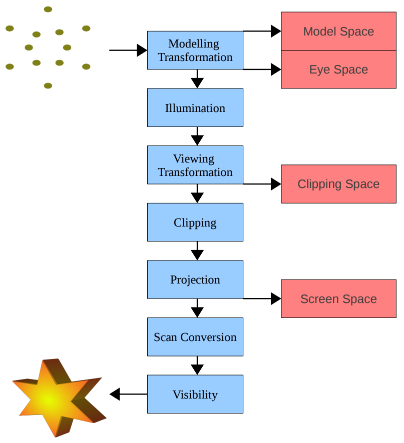


Fig. 3. Transformation Pipeline and Coordinate Systems.

thus mapping them to world coordinates. Then, the world coordinate system is transformed into a camera system, which privileges the viewer. These steps, taking model coordinates to world coordinates and then to camera coordinates, are usually applied in sequence.

Nowadays, the model's properties can be set in its vertices. Therefore, each vertex can have data, such as colour, normal and texture coordinates. In the Lightning Stage, each vertex colour is adjusted based on the light, considering its own position, its normal, the light source position, and other aspects.

The perspective projection defines a view frustum, which is a truncated pyramid (Fig. 4). The next stage maps the pyramid into a parallelepiped, as a result of applying the perspective transformation. On the other hand, the orthogonal projection does not need that step. Finally, each coordinate is then normalized to fit in the rande $[-1, 1]$.

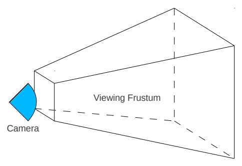


Fig. 4. View Frustum for a Perspective Projection.

In the Clipping Stage, primitives completely outside the view frustum are discarded. In some cases, the primitive can be partially inside the frustum, like the one in Fig. 5a, and the primitive must be divided (Fig. 5b).

In the Projection Stage, we use the screen space, which is a discrete space with the same resolution of the framebuffer. Therefore, the video card projects each vertex coordinate (x,y) in the screen. After this stage, the primitives have integer vertex coordinates (x,y) in the screen space and a z value representing the vertex depth.

The next step is to fill the primitive. In order to do so, each vertex attribute is interpolated, in each fragment corresponding to a pixel in the framebuffer. The z-depth is also interpolated and stored in the z-buffer. This process is called scan conversion. In each interpolated fragment, a decision about drawing it or not, in the corresponding pixel, is made according to the value stored in the z-buffer. If the currently evaluated fragment depth is smaller than the value already stored in the z-buffer, then the z-buffer is updated. Otherwise, the fragment is discarded.

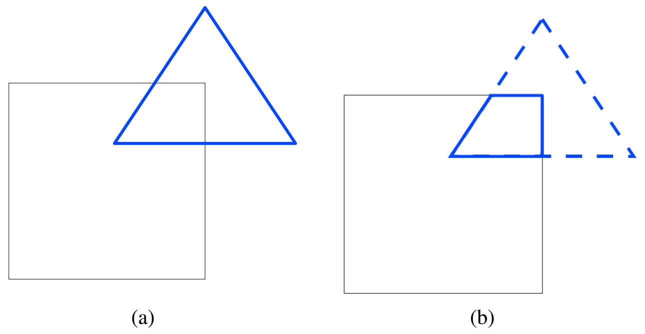


Fig. 5. Primitive clipping, so it is completely inside the view frustum. (b) Result of clipping the primitive in (a)

After all primitives have been processed, the scene is completely rendered in the framebuffer, which is now available for screen presentation.

Further details about this process can be found in OpenGL [13].

B. Vertex and Fragment Shaders

We have pointed that the fixed-function pipeline in hardware boosted the processing, but removed some flexibility. Video card manufacturers, on the other hand, have developed more complex architectures, which give back some functionality for programming these stages. Therefore, the Shaders were born, as graphic pipelines programmable stages, allowing the transfer of the model, scene data, and code, for replacing some of the fixed-functions. The first stages created were the Vertex and Fragment Shaders, and Fig. 6 presents some of the previous stages being replaced for these shaders. The input data can be manipulated in the Vertex Shader, and the final image can be customized in the fragment Shader. Each stage, including Geometry and Tessellation, will be further explained in the sequel.

C. Geometry Shader

Unlike the Vertex Shader, which replaces a piece of the graphics pipeline, or the Fragment Shader, the Geometry Shader is a new stage included into the programmable pipeline. It is like a Vertex Shader extension, capable of analysing a whole primitive, instead of seeing only one vertex. This way, it is possible to know some vertex neighbours, and produce new primitives.

D. Tessellation Shaders

Tessellation is composed by three new stages, with two new shader types: Tessellation Control Shader, Tesselator (non

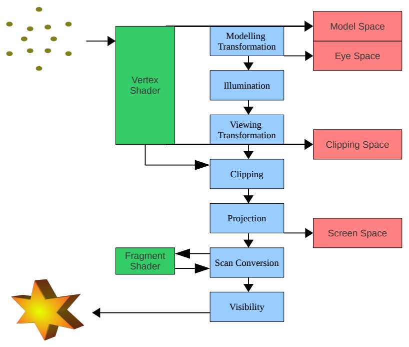


Fig. 6. Transformation Pipeline with Vertex and Fragment Stages.

programmable), and Tesselation Evaluation Shader. They are applied after the vertex stage and before de geometry shader stage: V (T_c T T_e) G F

Instead of simply displacing vertices, now the pipeline can also create or delete vertices before the Clipping Stage. Some vertices can be grouped working as controllers, called a patch, which can produce a new surface, for instance, a Bézier surface patch.

IV. GLSL LANGUAGE

A. *Replacing the fixed-function pipeline, first version (vertex shader)*

```
void main()
{
  vec4 ecPos = gl_ModelViewMatrix * gl_Vertex;

  vec4 amb = gl_Color * 0.2;

  gl_FrontColor = amb;
  gl_FrontColor.a = 1.0;

  gl_Position = gl_ProjectionMatrix * ecPos;
}
```

Listing 1. VertexCode: Replacement of the fixed-function pipeline

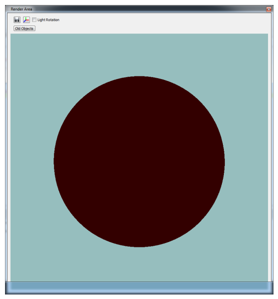


Fig. 7. Application of 1 onto a sphere, by using ShaderLabs.

The Vertex Shader can be understood by looking at listing 1, and observing the corresponding output in Fig. 7. The same output can be easily achieved by copying the code into ShaderLabs, which can guide and help a user to learn the shader. The code does not employ any custom technique, and its effect just mimics what has already been done in the fixed-function pipeline. Nonetheless, it will be useful in understanding several concepts.

In Fig. 6 it is depicted a scheme where the Vertex Shader replaces the steps of modelling, lightning and viewing. In VertexCode in Listing 1, the modelling corresponds to the first line of function *main*

```
vec4 ecPos = gl_ModelViewMatrix * gl_Vertex;
```

where the new variable *ecPos* holds the position of the eye vertex. This transformation is applied by multiplying the matrix *gl_ModelViewMatrix* and the vertex position in *gl_Vertex*. Since *gl_Vertex* is a four-dimensional vector, and *gl_ModelViewMatrix* a 4×4 matrix, the result is a four-dimensional vector, thus defining the type *vec4* of variable *ecPos*. GLSL allows the definition of variables of several pre-defined types, such as vectors of dimension 2, 3 or 4, and matrices (square or not) of at most four dimensions. To access each vector position, we can use the notation in *[]*, where the first position corresponds to index 0 (zero). Alternatively, we can also use *.x*, *.y*, *.z*, *.w*, for the first, second, third and fourth positions. A vertex is represented as a four-dimensional vector in *homogeneous coordinates*. The fourth coordinate allows the representation of a translation as a 4×4 matrix, for instance. The later transformation is a transformation in the *affine space*. The matrix type follows the notation *matI*, where I can be replaced by either 2, 3 or 4, for square matrices, or *matIxJ* for matrices $J \times I$, that is, J lines and I columns. Furthermore, the operator * defines a multiplication of a matrix by a vector, or a matrix by a matrix.

The viewing step corresponds to the last line of function *main*

```
gl_Position = gl_ProjectionMatrix * ecPos;
```

where a vertex, in eye coordinate, is multiplied by the matrix *gl_ProjectionMatrix*. This matrix maps a vertex in eye space to a vertex in clipping space. These transformations can be executed in a single step

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

or

```
gl_Position = ftransform();
```

which produces the same result. It should be noted that the the Vertex Shader must hold the vertex position, in clipping coordinates, in the output variable *gl_Position*. The Vertex Shader also holds, in the output variables *gl_FrontColor* and *gl_BackColor*, the vertex colour, after illumination, taking into account the front and back faces, respectively. In VertexCode in Listing 1, only the ambient illumination component is applied to the front face.

```
vec4 amb = gl_Color * 0.2;
gl_FrontColor = amb;
gl_FrontColor.a = 1.0;
```

The *alpha* channel of the final colour was set to 1.0, because in this example, we did not want any transparency.

B. Replacing the fixed-function pipeline, second version (vertex shader)

```
1  void PointLight(in vec3 ecPosition3,
2                  in vec3 N,
3                  in float shininess,
4                  inout vec4 diffuse,
5                  inout vec4 specular)
6  {
7
8    vec3  L;
9    L = vec3(gl_LightSource[0].position) - ecPosition3;
10   L = normalize(L);
11   N = normalize(N);
12
13   float diffAtt = max(0.0, dot(N, L));
14
15   float specAtt = 0.0;
16   if(diffAtt > 0.0)
17   {
18     vec3 R = reflect(-L, N);
19     vec3 V = normalize(-ecPosition3);
20
21     specAtt = max(0.0, dot(R, V));
22     specAtt = pow(specAtt, shininess);
23   }
24
25   diffuse  = diffuse  * diffAtt;
26   specular = specular * specAtt;
27 }
28
29 void main()
30 {
31   vec4 ecPos = gl_ModelViewMatrix * gl_Vertex;
32   vec3 ecPos3 = vec3(ecPos)/ecPos.w;
33   vec3 n = gl_NormalMatrix * gl_Normal;
34
35   vec4 amb = gl_Color * 0.2;
36   vec4 dif = gl_Color;
37   vec4 esp = vec4(1.0);
38
39   PointLight(ecPos.xyz, n, 16.0, dif, esp);
40
41   gl_FrontColor = amb + dif + esp;
42   gl_FrontColor.a = 1.0;
43   gl_Position = gl_ProjectionMatrix * ecPos;
44 }
```

Listing 2. VertexCode: Replacement of the fixed-function pipeline

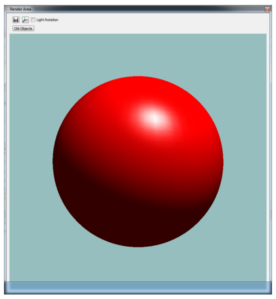


Fig. 8. Application of VertexCode in Listing 2 onto a sphere, by using ShaderLabs.

In VertexCode in Listing 1 there is no illumination. Nonetheless, is is supposed that the vertex shader perform the vertex illumination coherently. VertexCode in Listing 2 depicts a model of *shading* that can be applied in the vertex shader. In the example, the final colour is obtained by means of the ambient, diffuse and specular illumination components. Each of theses three variables, plus variable *gl_FrontColor*, are of type vec4. Its components can be accessed as *.r*, *.g*, *.b* and *.a*, as discussed before. GLSL considers an alpha channel value as being between 0.0 and 1.0, and values outside this range can be interpreted differently by a video card.

Most of the code calculate illumination components. This computation is performed by the function *PointLight*, on line 1 of VertexCode in Listing 2, which returns no value (*void*). However, the parameters can be *in*, *out* or *inout*. When a parameter does not have a modifier, or is *in*, then the variable is local, and brings information to the function, that is, any new value attributed to the variable will be lost when leaving the function. When a variable is *inout*, it also brings information to the function, but new values persist after the end of the function. Variables *out* do not bring information to the function, but new values are kept after the end of the function. Modifiers *out* and *inout* are useful when returning more than one piece of information in the same function. The function that computes the diffuse and specular illumination components return them in the variables *diffuse* and *specular*, respectively.

The computation of the components, in the equation below, is a simplification of the Phong reflection model[14].

$$C_p = K_a + K_d(N \cdot L) + K_s(R \cdot V)^n$$

In the equation, all vectors (N, L, R and V) are normalized: N is the vertex normal, L is the vector pointing from the vertex to the light, R is the reflection of vector L about the vertex normal, and V is the vector from the vertex to the eye. K_a, K_d and K_s are the ambient, diffuse and specular colour components, before attenuation. Factor n is called *shininess*, and it is a constant defined by the material.

The light sources in GLSL are kept in the array *gl_LightSource*. Since this example possesses only a single light source, only the array index 0 is accessed. Each array entry is a structure of type *gl_LightSourceParameters*, which keeps, among other attributes, the light source position, in eye space.

GLSL has several functions, including some for performing vector calculation, used during the illumination step, for example: the function *normalize* returns a normalized vector. Since operator $*$ could be ambiguous, meaning either dot or vector product, GLSL supplies two functions: *dot*, which returns a float holding the dot product between two vectors, and *cross*, which returns the vector product. Function *reflect*, returns the reflection vector [1] of the first parameter vector about the second parameter vector (supposed normalized). Since illumination is performed in eye space, a vector from the vertex to the eye would be $o - v$. Also, because o is null, $(0, 0, 0)$, then V is just $-v$.

A vertex shader code is executed in parallel for all vertices of the model. Therefore, the input for a vertex shader are vertices, and its attributes, such as colour, normal, texture coordinates, etc. It should be noted that there is no edge in a vertex shader code, that is, there is no adjacency information.

[1]Vector L is a vector emanating from the light source. Therefore, is has to be inverted in VertexCode in Listing 2.

C. Replacing the fixed-function pipeline, third version (fragment shader)

```
void main()
{
  gl_FragColor = gl_Color;
}
```

Listing 3. FragmentCode: Replacement of the fixed-function pipeline

For rendering, it is necessary to interpolate the vertex attributes. In VertexCode in Listing 2, the only information interpolated is the vertex colour. As a consequence, because the colour is computed on a vertex and interpolated during the scan conversion, it is said this example uses the *Gouraud shading model*[15]. The code in FragmentCode in Listing 3 produces the same result depicted in Fig. 8, which means no change is performed. This is due to the fact that the fragment shader does not replace the graphic pipeline, but only improves it. This code shows that *gl_FragColor* is the main output of the fragment shader, where the fragment colour is set. A fragment is equivalent to a pixel, however, the fragment refers to a primitive that can be discarded for another primitive closer to the viewer, or merged with another fragment, when the *alfa blending* is activated. The input variable *gl_Color* receives the interpolated value from variables *gl_FrontColor* or *gl_BackColor*, set in the vertex shader, depending on whether the rendered primitive is facing front or back. Both variables shown in the example are of type *vec4*.

D. Phong and Texture

```
varying vec3 N;
varying vec3 L;
varying vec3 V;

void main()
{
  vec4 ecPos = gl_ModelViewMatrix * gl_Vertex;
  vec3 ecPos3 = vec3(ecPos)/ecPos.w;

  L = vec3(gl_LightSource[0].position) - ecPos3;
  N = gl_NormalMatrix * gl_Normal;
  V = -ecPos3;

  gl_FrontColor = gl_Color;
  gl_Position = gl_ProjectionMatrix * ecPos;
}
```

Listing 4. VertexCode: Phong Shading

```
varying vec3 N;
varying vec3 L;
varying vec3 V;

void main()
{
  vec3 n = normalize(N);
  vec3 l = normalize(L);
  vec3 v = normalize(V);

  vec4 color = gl_Color;

  float diffAtt = max(0.0, dot(n, l));
  float specAtt = 0.0;
  if(diffAtt > 0.0)
  {
      vec3 r = reflect(-l, n);

      specAtt = max(0.0, dot(r, v));
      specAtt = pow(specAtt, 16.0);
  }
  vec3 amb = vec3(color)*0.2;
  vec3 dif = vec3(color)*diffAtt;
  vec3 spe = vec3(1.0)*specAtt;

  gl_FragColor = vec4(amb+dif+spe, 1.0);
}
```

Listing 5. FragmentCode: Phong Shading

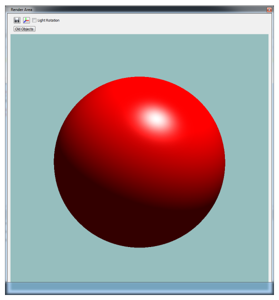


Fig. 9. Application of VertexCode in Listing 4 and FragmentCode in Listing 5 onto a sphere, by using ShaderLabs.

The application of VertexCode in Listing 4 and FragmentCode in Listing 5 can be seen in Fig. 9. Although very similar to Fig. 8, the *Phong Shading* produces a better quality rendering, which can be seen in Fig. 10.

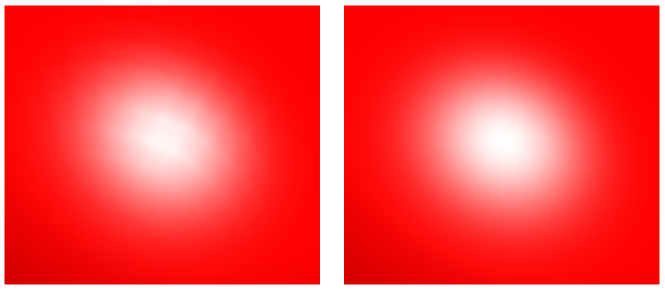


Fig. 10. *Gouraud Shading* on the left versus *Phong Shading* on the right.

In the example shown on FragmentCode in Listing 5, the illumination computation is postponed to the scan conversion phase, by interpolating in each fragment the vectors needed to compute the illumination, instead of interpolating the colours already shaded. For passing vertex interpolated data to the fragment, it is enough that both shaders have variables with the same name and type, globally declared as *varying*.

```
varying vec3 N;
varying vec3 L;
varying vec3 V;
```

Because interpolation changes vector magnitudes, it is common to normalize the vectors in the fragment shader only.

```
vec3 n = normalize(N);
vec3 l = normalize(L);
vec3 v = normalize(V);
```

Similarly to the vertex shader, the fragment shader code is also executed in parallel in all fragments, without checking any other fragment, and not changing its position.

Texturing is also performed in the fragment shader, by passing the interpolated texture coordinates, from the vertex shader to the fragment shader, using a variable set as *varying*. Therefore, texture can be applied by using a global variable in both codes,

```
varying vec2 texCoord;
```

and including somewhere, in function main of the vertex shader, the following line:

```
texCoord = gl_MultiTexCoord0.xy;
```

For texturing, each vertex of the model must have a n-dimensional texture coordinate, but generally, texture is two-dimensional. In the 2D case, the first two coordinates of variable *gl_MultiTexCoord0* hold the vertex texture coordinates, and each texture coordinate should be normalized in the range [0,1]. The mapping is shown in Fig. 11, where coordinate $(0,0)$ is the texture lower left corner, and coordinate $(1,1)$ the upper right corner.

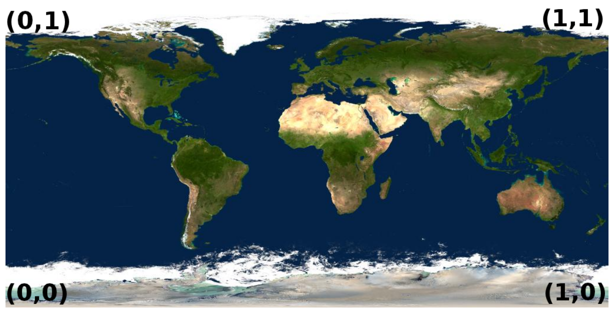


Fig. 11. Texture mapping.

In the fragment shader, it is necessary to use a global variable:

```
uniform sampler2D sampler2d0;
```

This variable is set to *uniform*, which means its value does not change during the processing. Variables of type *sampler2D*, representing a 2D texture, are always *uniform*.

The goal of this work is not show how to unwrap a texture and send it to the shaders. However, it is possible to accomplish that in a simple way, by using ShaderLabs. For this purpose, a texture is chosen with a single mouse click, and it will be available in a variable, whose name is visible beside the imported texture miniature. In general, the software sets names of the form *sampler2d***X** to all textures, where *X* is a number depending on the order in which the texture was inserted, as shown in the example.

To finish the texture, one should replace in Fragment-Code in Listing 5, the line:

```
vec4 color = gl_Color;
```

for

```
vec4 color = texture2D(sampler2d0, texCoord);
```

Therefore, instead of getting the base colour from the colour coming from the vertex shader, it will get the base texture colour using the function *texture2D*. This function returns the texture colour, whose identifier is passed as the first parameter, in the coordinate passed as the second parameter.

E. Geometry pass-through and Spike

```
#version 120
#extension GL_ARB_geometry_shader4 : enable

void main()
{
  for(int i = 0; i < gl_VerticesIn; ++i)
  {
    gl_FrontColor = gl_FrontColorIn[i];
    gl_Position = gl_PositionIn[i];
    EmitVertex();
  }
  EndPrimitive();
}
```

Listing 6. VertexCode: Keeping original pipeline

Including GeometryCode in Listing 6 and VertexCode in Listing 2, produces the result shown in Fig. 8. The reason is that the code just passes forward, vertices already processed in the vertex shader, without any modification.

```
#version 120
#extension GL_ARB_geometry_shader4 : enable
```

The code above is responsible for configuring which GSL version will be used.

In the geometry shader, the processing is performed for each primitive. The primitive type (triangles, lines, points, etc) should be set in setup application.

The geometry shader outputs the same information as the vertex shader, that is, the result is the vertex colour and position in clipping coordinates. Since nothing has been changed in the example, each vertex is set with what has already been set before.

```
gl_FrontColor = gl_FrontColorIn[i];
gl_Position = gl_PositionIn[i];
```

Variables ending with *In* are input variables, which represent data configured as output in the vertex shader. They are arrays, because a primitive may have more than one vertex.

Every time a vertex is set, it needs to be passed along, by using the function *EmitVertex*. When all vertices of a primitive have already been set, it is necessary to signal that the primitive is finished, by using the function *EndPrimitive*. In the given example, either the input primitive or the output primitive are triangles. However, this is not always this way.

```
#version 120
#extension GL_ARB_geometry_shader4 : enable
varying in vec3 NIn[];
varying out vec3 N;
void main()
{
  vec4 p[3];
  p[0] = gl_PositionIn[0];
  p[1] = gl_PositionIn[1];
  p[2] = gl_PositionIn[2];

  vec3 n = cross(vec3(p[2]-p[1]), vec3(p[0]-p[1]))*5;
  vec4 c = (p[0]+p[1]+p[2])/3.0;
  vec4 piv = vec4(c.xyz + n, 1.0);
  for(int i = 0; i < 3; ++i)
  {
    N = cross(vec3(p[i]-piv), vec3(p[(i+1)%3]-piv));
    gl_FrontColor = gl_FrontColorIn[i];
    gl_Position = gl_ProjectionMatrix * piv;
    EmitVertex();
    gl_Position = gl_ProjectionMatrix * gl_PositionIn[
        i];
    gl_FrontColor = gl_FrontColorIn[i];
    EmitVertex();
    gl_Position=gl_ProjectionMatrix*gl_PositionIn[(i
        +1)%3];
    gl_FrontColor = gl_FrontColorIn[(i+1)%3];
    EmitVertex();

    EndPrimitive();
  }
}
```

Listing 7. GeometryCode: Spike

In the GeometryCode in Listing 7 example, the geometry shader is discarding the initial triangle, given as input, and creating three new triangles based on the original triangle. The rationale is creating a new vertex on the triangle centroid, and raising it in the direction of its normal, thus creating a ridge (see Fig. 12). For this purpose, it was configured, in the vertex shader, a variable *gl_Position* with the vertex position in eye coordinates, and left to the geometry shader the mapping of the vertex to clipping coordinates.

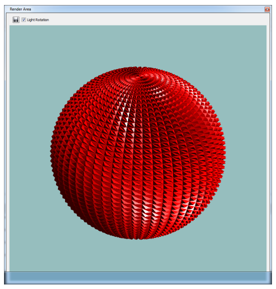


Fig. 12. Application of GeometryCode in Listing 7 onto a sphere.

F. Simple height-map (tessellation shaders)

The main purpose of the tessellator is to subdivide a primitive. The next example subdivides four triangles to create a terrain. A terrain needs a reasonable geometry resolution (Fig. 13), and TessCtrlCode in Listing 8 and TessEvalCode in Listing 9 will increase the resolution, by subdividing the triangles.

```
#version 400
layout(vertices = 3) out;

// tessellation value indicates how many subdivisions
const float tess_level = 4;

void main() {
  #define id gl_InvocationID
  // simply transfering gl_Position from input to
      output
  gl_out[id].gl_Position = gl_in[id].gl_Position;

  if( id == 0 ) {
  gl_TessLevelOuter[0] = tess_level;
    gl_TessLevelOuter[1] = tess_level;
    gl_TessLevelOuter[2] = tess_level;

    gl_TessLevelInner[0] = tess_level;
  }
}
```

Listing 8. TessCtrlCode: Triangle Subdivision.

```
#version 400 compatibility
//input: triangles, made by ccw triangles
//(equal_spacing) means that
//the squares are all equal sized
layout(triangles, equal_spacing, ccw) in;

void main(){
//each vertex in the subdivided triangle
//is assigned a barycentric (u,v,w) coordinate based
    on its location

    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    float w = gl_TessCoord.z;

    vec3 p0 = gl_in[0].gl_Position.xyz;
    vec3 p1 = gl_in[1].gl_Position.xyz;
    vec3 p2 = gl_in[2].gl_Position.xyz;

//model position relative to the three vertices of the
      outer triangle
   vec3 model_position = p0*u + p1*v + p2*w;

//projects to screen coordinates
    gl_Position = gl_ModelViewProjectionMatrix * vec4(
          model_position, 1.0 );
}
```

Listing 9. TessEvalCode: Triangle Subdivision.

The TessCtrlCode in Listing 8 and TessEvalCode in Listing 9 perform the triangle subdivision, as can be seen in Fig. 14;

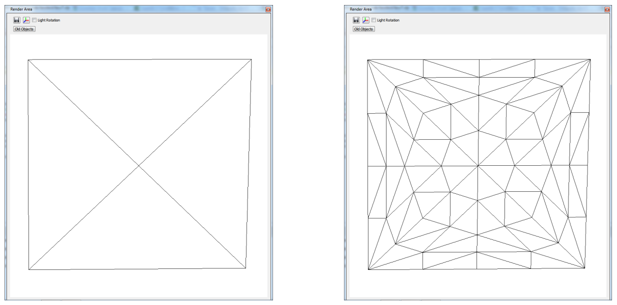


Fig. 14. The original primitive on left and the result, by executing a tessellation stage

Tessellation control shaders transform an input patch specified by the application, computing per-vertex and per-patch attributes for a new output patch. A fixed-function tessellation primitive generator subdivides the patch, and tessellation evaluation shaders are used to compute the position and attributes of each vertex produced by the tessellator.

The tessellation primitive generator then decomposes a patch into a new set of primitives using the tessellation levels to determine how finely tessellated the output should be. The primitive generator begins with either a triangle or a quad, and splits each outer edge of the primitive into a number of segments approximately equal to the corresponding element of the outer tessellation level array (`gl_TessLevelOuter`) . The interior of the primitive is tessellated according to elements of the inner tessellation level array (`gl_TessLevelInner`).

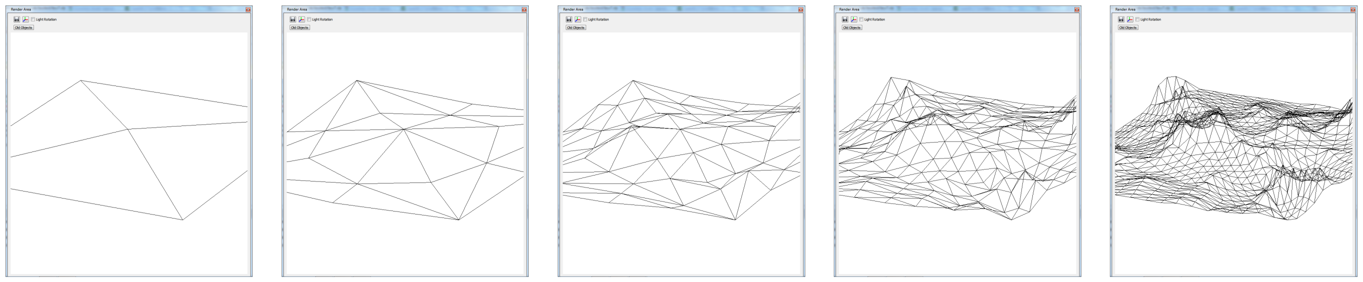


Fig. 13. A sequence of increasing resolutions. From left to right: original primitives, and subdivisions with factor 2, 4, 8, and 16

For each vertex produced by the tessellation primitive generator, the tessellation evaluation shader is run to compute its position and other attributes of the vertex, using its (u,v) or (u,v,w) coordinates.

Finally, by setting the z coordinate of the vertices of the model, defines a terrain onto the square (Fig. 14). For visualizing the terrain, any previously loaded texture can be used as a height map. Therefore, adding the following piece of code in TessEvalCode in Listing 9 produces a simple terrain visualizer Fig. 13:

```

uniform sampler2D sampler2d0;

// compute the final texture coordinate
vec2 tc = texCoord[0]*u + texCoord[1]*v + texCoord[2]*w;

// get the height factor from texture
float height = texture2D( sampler2d0 , tc ).r;

// and change the z value
model_position.z = height;
```

Listing 10. TessEvalCode: Terrain Visualization.

V. Shader Applications

In this section we introduce some complete examples using GLSL shaders. Each example focus on different programmable stages.

- Vertex Noise
- Fragment Ray-casting
- Geometry Hair
- Tessellation LOD

A. *Vertex Noise*

This application is an example of an intense vertex shader use. The input is a sphere on which the shader is applied. Based on Perlin's Noise [16], for each vertex of a sphere, a noise is included before computing its final location (Fig. 10). The core computation is done in the vertex shader.

```
#define B  32       // table size
#define B2 66       // B*2 + 2
#define BR 0.03125  // 1 / B

float s_curve(float t)
{     return t*t*(3.0-2.0*t);}

// 3D version
float noise(vec3 v, vec4 pg[]){...}
// 2D version
float noise(vec2 v, vec4 pg[]){...}
// 1D version
float noise(float v, vec4 pg[])
{
      v = v + 10000.0;

      float i = fract(v * BR) * float(B);    // index [0, B-1]
      float f = fract(v);                   // fractional position

      // compute dot products between gradients and vectors
      vec2 r;
      r[0] = pg[int(i)].x * f;
      r[1] = pg[int(i) + 1].x * (f - 1.0);

      // interpolate
      f = s_curve(f);
      return mix( r[0], r[1], f);
}

const float Displacement = 2.0;
uniform vec4 pg[B2];            // permutation/gradient table

void main()
{
      vec4 noisePos = gl_TextureMatrix[0] * gl_Vertex;
      float i = (noise(noisePos.xyz, pg) + 1.0) * 0.5;
      gl_FrontColor = vec4(i, i, i, 1.0);
      // displacement along normal
      vec4 position = gl_Vertex +
            (vec4(gl_Normal, 1.0) * i * Displacement);
      position.w = 1.0;
      gl_Position = gl_ModelViewProjectionMatrix *
            position;
}
```

Listing 11. VertexCode: Perlin Noise (from NVidia Code Sample [17])

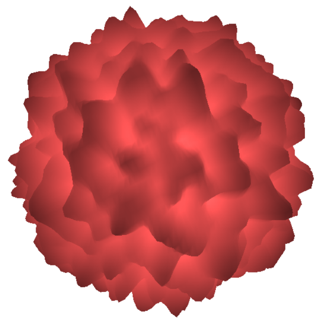


Fig. 15. Perlin noise applied on the vertices of a sphere.

The code in VertexCode in Listing 11 is based on Perlin's original code [17], where:

- `Displacement` is a variable that sets the maximum displacement of a vertex from its original position in the normal direction (the original radius is 1).
- `mix` performs a linear interpolation between two values.
- `fract` returns the fractional part of the argument.
- `pg` is a 32*4 floating-point table that contains the noise permutation gradient table, which is set by the application and passed to the shaders through a `uniform` variable.

B. Fragment-shader, Sphere ray-casting

```
varying vec3 vv;
void main()
{
  vec4 eye = gl_ModelViewMatrixInverse *
        vec4(0.0, 0.0, 0.0, 1.0);
  vv = vec3(gl_Vertex - eye);
  gl_FrontColor = gl_Color;
  gl_Position = ftransform();
}
```

Listing 12. VertexCode: Ray Casting

```
varying vec3 vv;
const float radius = 1.0;

vec3 shading(vec3 N, vec3 L, vec3 R,
    vec3 V, float n, vec3 baseColor)
{
  float diff = max(0.0, dot(N,L));
  float spec = 0.0;
  if(diff > 0.0)
    spec = pow(max(0.0,dot(R,V)), n);
  return (baseColor * 0.2) + baseColor * diff + spec;
}
void main()
{
  vec3 eye = vec3(gl_ModelViewMatrixInverse *
                        vec4(0.0, 0.0, 0.0, 1.0));
  float a = dot(vv,vv);
  float b = dot(vv,eye);
  float c = dot(eye,eye) - radius*radius;

  float delta = b*b - a*c;
  if(delta < 0.0)
    discard;
  delta = sqrt(delta);
  float t = (-b - delta)/a;

  vec4 p = vec4(eye + t*vv, 1.0);
  vec3 n = normalize(p.xyz);
  vec4 lp = gl_ModelViewMatrixInverse *
                gl_LightSource[0].position;
  vec3 l = normalize(vec3(lp - p));

  gl_FragColor.rgb = shading(n, l, reflect(-l, n),
                normalize(-vv.xyz), 16.0, gl_Color.
                    rgb);
  gl_FragColor.a = 1.0;

  p = gl_ModelViewProjectionMatrix * p;
  gl_FragDepth = ((gl_DepthRange.far - gl_DepthRange.
        near) *
                p.z/p.w +
                gl_DepthRange.near + gl_DepthRange.far)
                    /2.0;
}
```

Listing 13. FragmentCode: Ray Casting

In 2004, Toledo and Levy[18] introduced a new way of extending the graphics pipeline. They proposed a technique to render implicit primitives in GPU, by using a ray casting algorithm in the fragment shader. The example in VertexCode in Listing 12 and FragmentCode in Listing 13 renders a sphere inside a box. The primitive sent from CPU to GPU was a cube, and it can be performed by choosing a cube primitive on ShaderLabs.

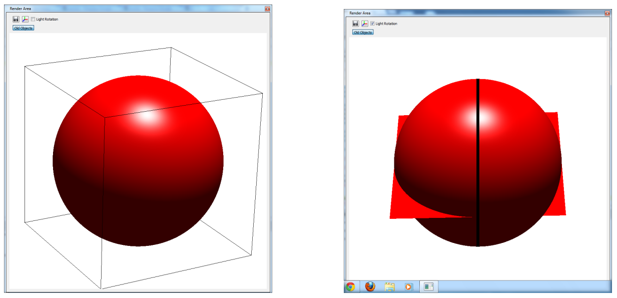


Fig. 16. *Left*: application of VertexCode in Listing 12 and FragmentCode in Listing 13 onto a cube, by using ShaderLabs. *Right*: Correct fragment depth on the left versus a depth obtained using the face of the cube, on the right.

In this example, a ray is cast from the camera origin to each vertex in model coordinates:

```
  vec4 eye = gl_ModelViewMatrixInverse *vec4(0.0, 0.0,
        0.0, 1.0);
  vv = vec3(gl_Vertex - eye);
```

Listing 14. FragmentCode: Model to eye transformation.

and stored in the *vv* variable. The *gl_ModelViewMatrixInverse* inverts the *gl_ModelViewMatrix*, which provides a transformation from eye coordinates to model coordinates. Since the eye position in eye coordinates is $(0, 0, 0)$, the first line computes the eye position in model coordinates, while the second line computes the ray direction. An intersection equation must be solved, for each fragment, in order to get the intersection point between a parametrized ray and an implicit sphere.

Given a parametrized ray $r(t) = P_0 + t\vec{d}$, where P_0 is the initial point, $\vec{d}$ is a base vector that gives the ray direction; and the sphere implicit equation, $x^2+y^2+z^2 = r^2$; the intersection equation is:

$$t^2(\vec{d} \cdot \vec{d}) + 2t(\vec{d} \cdot P_0) + (P_0 \cdot P_0) - r^2 = 0$$

If *delta* < 0, then the fragment is discarded with the fragment command *discard*.

The last line in FragmentCode in Listing 13 computes the correct depth value of the fragment. This is important since the original fragment depth is the one of the face of the cube. This depth equation can be found in The OpenGL Specification [19]. A comparison between sphere depth and cube depth is shown in Fig. 16.

The first line, of the following code, computes p in clipping coordinates ($p_c = (xw, yw, zw, w)$). However, the depth equation expects the z value in screen coordinates ($p_d = (x, y, z, 1)$). Therefore, $p.z$ is divided by $p.w$.

```
p = gl_ModelViewProjectionMatrix * p;
gl_FragDepth = ((gl_DepthRange.far -
                gl_DepthRange.near) * p.z/p.w +
                gl_DepthRange.near + gl_DepthRange.far)
                    /2.0;
```

Listing 15. FragmentCode: Fragment depth value.

C. Geometry Hair

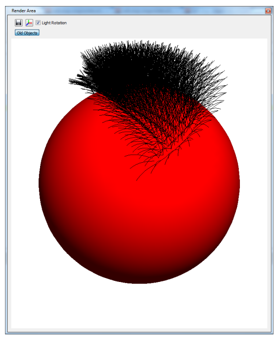


Fig. 17. Application of VertexCode in Listing 16 and GeometryCode in Listing 17 onto a sphere, by using ShaderLabs.

```
varying vec3 NIn;
void main()
{
  NIn = gl_Normal;
  gl_Position = gl_Vertex;
}
```

Listing 16. VertexCode: Mohican Hair

```
#version 120
#extension GL_EXT_geometry_shader4: enable
varying in vec3 NIn[];
const vec4 black = vec4(0.0, 0.0, 0.0, 1.0);
const float uDroop = 1.0;
const int uLength = 8;
const float uStep = 0.3;
const int numLayers = 2;
vec3 N0, N01, N02;
vec4 V0, V01, V02;
vec4 vs[3];
vec3 ns[3];
void produceVertices(float s, float t)
{
  vec4 v = V0 + s*V01 + t*V02;
  vec3 n = normalize(N0 + s* N01 + t*N02)*0.2;
  for(int i = 0; i <= uLength; ++i)
  {
    gl_Position = gl_ProjectionMatrix * v;
    gl_FrontColor = black;
    EmitVertex();
    v.xyz += uStep * n;
    float dec = i*0.02;
    v.y -= uDroop * float(dec*dec);
  }
  EndPrimitive();
}

void setup()
{
  for(int i = 0; i < 3; ++i)
  {
     vs[i] = gl_ModelViewMatrix * gl_PositionIn[i];
     ns[i] = gl_NormalMatrix * NIn[i];
  }
  V0 = vs[0];
  V01 = vs[1] - vs[0];
  V02 = vs[2] - vs[0];
  if(dot(ns[0], ns[1]) < 0.0)
    ns[1] = -ns[1];
  if(dot(ns[0], ns[2]) < 0.0)
    ns[2] = -ns[2];
  N0  = normalize(ns[0]);
  N01 = normalize(ns[1] - ns[0]);
  N02 = normalize(ns[2] - ns[0]);
}
void main()
{
  if(gl_PositionIn[0].x > 0.1 &&
     gl_PositionIn[1].x > 0.1 &&
     gl_PositionIn[2].x > 0.1 &&
     gl_PositionIn[0].z > 0.5 &&
     gl_PositionIn[1].z > 0.5 &&
     gl_PositionIn[2].z > 0.5 )
  {
    setup();
    float dt = 1.0/numLayers;
    float t = 1.0;
    for(int it = 0; it <= numLayers; ++it)
    {
      float smax = 1.0 - t;
      int nums = it + 1;
      float ds = smax/float(nums-1);
      float s = 0.0;
      for(int is = 0; is < nums; ++is)
      {
        produceVertices(s,t);
        s += ds;
      }
      t -= dt;
    }
  }
}
```

Listing 17. GeometryCode: Mohican Hair

The mohican hair effect shown in Fig. 17 is a good geometry shader example. The result was obtained by executing the VertexCode in Listing 16 and GeometryCode in Listing 17 onto a sphere, by ShaderLabs. In this example, the input primitive is a triangle and the output primitive is a line-strip.

The VertexCode in Listing 16 is a simple code to pass the vertex position and normal, in model coordinates, to the geometry shader. The first line of the hair GeometryCode in Listing 17 *main function* is responsible for selecting the appropriate piece of the sphere, and discarding the other part. In order to select the appropriate part, the vertex must be in model coordinates. Before explaining the code, it is important to understand the triangle parametrization used in this shader.

Given three non-collinear vertices, v_0, v_1 e v_2, it is possible to generate a vertex into the triangle, by using an affine combination:

$$v(s,t) = v_0 + s(v_1 - v_0) + t * (v_2 - v_0) \qquad (1)$$

Therefore, it is simpler to understand the *produceVertices* function. It receives two parameters (s and t) and applies the affine combination to compute a vertex and normal inside the triangle, where $V01 = V1 - V0$ and $V02 = V2 - V0$. Then, it produces *uLength* adjacent lines following the normal direction, with a small quadratic decay (`v.y -= uDroop * float(dec*dec)`), to simulate the gravity's effect on a hair strand.

The *setup* function performs the setup of some variables on eye coordinate. The *main* function computes the parameters of Eq. 1, in order to maximize the number of hair strands per face.

Because lines do not have a uniquely defined normal, illumination is not computed. However, it can be done using Mallo's approach[20]. This example was borrowed and adapted from Bailey [21].

D. Tessellation LOD (Level of Detail)

Terrain visualization is a common and relevant application, and for a realistic visualization, the corresponding surface mesh may have a large number of vertices.

As a consequence, for an efficient rendering, optimizations, such as a progressive level of detail, can be applied based on the distance to the viewer and the clipping plane position.

```

// function for returning the projected position
// of a given point
vec4 project(vec4 vertex){
    vec4 result = gl_ModelViewProjectionMatrix * vertex;
    result /= result.w;
    return result;
}

// function for converting a normal vector
// in device space to screen space
vec2 screen_space(vec4 vertex){
    return (clamp(vertex.xy, -1.3, 1.3)+1) * (wsize*0.5);
}

float level(vec2 v0, vec2 v1){
     return clamp(distance(v0, v1)/lod_factor, 1, 64);
}

bool offscreen(vec4 vertex){
  // tests if it's behind the camera
    if(vertex.z < -0.5){
        return true;
    }
  // checks the XY borders
    return any(
        lessThan(vertex.xy, vec2(-1.7)) ||
        greaterThan(vertex.xy, vec2(1.7))
    );
}
```

Listing 18. TessCtrlCode: Auxiliary code to listing 20.

For finding the best level of detail, in tess amount, listing 18 divides the distance by a given constant factor. The values are clamped in the range [1,64], which defines the subdivision level. Float values are valid, because the tesselation spacing can be fractional.

In listing 19, it is presented a clipping algorithm that returns whether the vertex is outside the screen. A True value means the vertex is outside, and therefore can be discarded.

```
layout(vertices = 3) out;

uniform vec2 wsize;
const float lod_factor = 10.0;

void main(){
  #define id gl_InvocationID
  // simply transferring gl_Position from input to output
  gl_out[id].gl_Position = gl_in[id].gl_Position;
  texCoord[id] = texCoordIn[id];

  // the following code (inside the if block)
  // will be executed only once per patch
  // HLSL has a similar concept, using ConstantHullShader
  if( id == 0 ) {
    // projected the 4 corner control points
    vec4 v0 = project(gl_in[0].gl_Position);
    vec4 v1 = project(gl_in[1].gl_Position);
    vec4 v2 = project(gl_in[2].gl_Position);

    if( all( bvec3(
      offscreen(v0),
      offscreen(v1),
      offscreen(v2)
    ))){
  // if all of them are outside the frustum,
  // the tess level is dropped to 0,
  // then no vertex will be produced by this patch
      gl_TessLevelInner[0] = 0;
      gl_TessLevelOuter[0] = 0;
      gl_TessLevelOuter[1] = 0;
      gl_TessLevelOuter[2] = 0;
    }
    else{
      // defining the tessellation factor for each edge
      vec2 ss0 = screen_space(v0);
      vec2 ss1 = screen_space(v1);
      vec2 ss2 = screen_space(v2);

      float e0 = level(ss1, ss2);
      float e1 = level(ss0, ss1);
      float e2 = level(ss2, ss0);

      // finally, assigns the chosen factors
      // internal tessellation level is mixed halfway
      // of the related opposite edges
      gl_TessLevelInner[0] = mix(e1, e2, 0.5);
      gl_TessLevelOuter[0] = e0;
      gl_TessLevelOuter[1] = e1;
      gl_TessLevelOuter[2] = e2;
    }
  }
}
```

Listing 19. TessCtrlCode: Terrain with LOD

The screen size is defined in the application, and the level of detail (LOD) factor means the desired quality of the scene, which is also configured by the application.

```
#version 400 compatibility
layout(triangles, fractional_odd_spacing, ccw) in;

// texture coordinates, to be used in the pixel shader
in vec2 texCoord[];

// depth value, to be used in the pixel shader
out float depth;

// texture with the height values
uniform sampler2D sampler2d0;

void main(){
  float u = gl_TessCoord.x;
  float v = gl_TessCoord.y;
  float w = gl_TessCoord.z;

  vec4 p0 = gl_in[0].gl_Position;
  vec4 p1 = gl_in[1].gl_Position;
  vec4 p2 = gl_in[2].gl_Position;

  vec4 point = p0*u + p1*v + p2*w;
  vec2 texC = texCoord[0]*u + texCoord[1]*v + texCoord[2]*w;

  float height = texture2D(sampler2d0, texC).r;
  point.z = height*0.4;

  // projects to screen coordinates
  gl_Position = gl_ModelViewProjectionMatrix * vec4(point);

  // simply copies as depth value
  depth = gl_Position.z;
}
```

Listing 20. TessEvalCode: Terrain with LOD

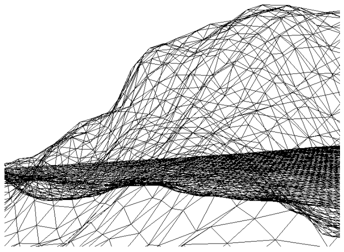


Fig. 18. Terrain visualization using Height Map in wireframe mode.

E. Sphere silhouette LOD

This last example also implements a LOD but to render a sphere, considering the silhouette as the target for a finner LOD. The input is an icosahedron geometry which was passed as patches from the OpenGL (see Listing 21) .

```
glPatchParameteri(GL_PATCH_VERTICES, 3);
glBegin(GL_PATCHES);
  glVertex3i(0,0,0);
  ...
glEnd();
```

Listing 21. Patches from the OpenGL

The output is a sphere that has a finner subdivision mesh in its border than in its interior (Fig. 19).

```
void main(void)
{
    gl_Position = gl_Vertex;
}
```

Listing 22. VertexCode: Icosahedron

```
1  #version 400 compatibility
2  layout( vertices = 3 )  out;
3
4  void main( )
5  {
6    #define id  gl_InvocationID
7
8    gl_out[ id ].gl_Position = gl_in[ id ].gl_Position;
9
10   if(id == 0)
11   {
12     vec3 N[4];
13     N[0] = (gl_in[ 1 ].gl_Position.xyz + gl_in[ 2 ].
           gl_Position.xyz)/2.0;
14     N[1] = (gl_in[ 0 ].gl_Position.xyz + gl_in[ 2 ].
           gl_Position.xyz)/2.0;
15     N[2] = (gl_in[ 0 ].gl_Position.xyz + gl_in[ 1 ].
           gl_Position.xyz)/2.0;
16     N[3] = (gl_in[ 0 ].gl_Position.xyz + gl_in[ 1 ].
           gl_Position.xyz + gl_in[ 2 ].gl_Position.xyz)
           /3.0;
17
18     for(int i = 0; i < 4; ++i)
19     {
20       N[i] = normalize(gl_NormalMatrix * N[i]);
21
22       float z = 1.0 - N[i].z * N[i].z;
23
24       if(i < 3 )
25         gl_TessLevelOuter[i] =  10.0 * z;
26       else
27         gl_TessLevelInner[0] =  10.0 * z;
28     }
29   }
30 }
```

Listing 23. TessCtrlCode: Icosahedron

```
1  #version 400 compatibility
2  layout( triangles , equal_spacing)  in;
3
4  void main( )
5  {
6    vec4 p0 = gl_in[0].gl_Position;
7    vec4 p1 = gl_in[1].gl_Position;
8    vec4 p2 = gl_in[2].gl_Position;
9
10   float u = gl_TessCoord.x;
11   float v = gl_TessCoord.y;
12   float w = gl_TessCoord.z;
13
14   vec3 point = (p0*u + p1*v + p2*w).xyz;
15   point = normalize(point.xyz);
16   gl_Position = gl_ModelViewProjectionMatrix * vec4(
         point, 1.0);
17 }
```

Listing 24. TessEvalCode: Icosahedron

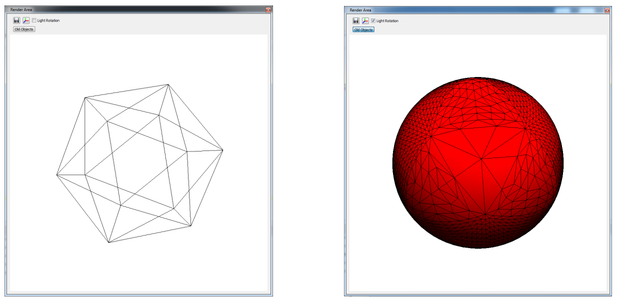


Fig. 19. Tessellated sphere from an icosahedron

Note that the `for` block between lines 18 and 28, in TessCtrlCode in Listing 23, sets the LOD resolution, according to the normal direction (line 23). Finally, the vertex final position must be projected to the sphere surface. In this example, this is computed in lines 14, 15 and 16 in (TessEvalCode in Listing 24), considering a zero-centered sphere of radius one.

VI. Conclusion

In this survey, we have selected a series of shader-based examples in a simple-to-complex order. For each one, the reader is invited to take a glance at the code before fully understanding it. This learning by example way is compatible with *Coding Dojo*, which is a recent group learning technique [1], [22]. Both are pull-systems for education (differently from traditional education, where the lessons are pushed to the audience, sometimes without respecting their time to assimilate the knowledge). This way, if the reader is willing to use this material in a class, we recommend that it should be based on *Coding Dojo*.

Another idea advocated in this survey is the use of a shader development environment to learn and practice GLSL language (or any shader language). This way, the practitioner avoids several issues, such as: environment configuration, CPU compilation or 3D model reading. Besides that, these environments provide almost instant result of their coding, accelerating the feedback time of code correctness. The recent development of Shaderlabs fills the gap of tools with all programmable stages. Shaderlabs is a concise tool and it was developed following Agile development practices [23]. More than that, we believe that it is possible to teach and learn visualization algorithms through shader programming, and there is no need to show them in the fixed-function pipeline.

A common complaining of those that program on shaders is the difficult in debugging their code. This is an issue, since the beginning of the programmable GPU, about ten years ago. Although some effort that has been done on this subject, we hope that in the near future there will be a practical solution for this problem, and it will probably be implemented as a feature in one of those shader tools.

Finally, this GLSL survey does not intend to be complete about the language itself, but it is a guide through examples. For further information about GLSL language, please check our list of references [3], [4], [5], [6], [8], [19], [21].

All examples shown in this paper can be found on [2].

REFERENCES

[1] C. Delgado, R. de Toledo, and V. Braganholo, "Uso de dojos no ensino superior de computação," in *XX Workshop sobre Educação em Computação (WEI), Anais do XXXII Congresso da Sociedade Brasileira de Computação (CSBC 2012)*, 2012.

[2] "Shaderlabs." [Online]. Available: http://www.dcc.ufrj.br/~shaderlabs/Shaderlabs

[3] R. J. Rost, *OpenGL(R) Shading Language*. Addison-Wesley Professional, Jan. 2006. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321334892

[4] R. Marroquim and A. Maximo, "Introduction to gpu programming with glsl," in *Computer Graphics and Image Processing (SIBGRAPI TUTORIALS), 2009 Tutorials of the XXII Brazilian Symposium on*, oct. 2009, pp. 3 –16.

[5] "Glsl: An introduction." [Online]. Available: http://nehe.gamedev.net/article/glsl_an_introduction/25007/

[6] "Glsl tutorial von lighthouse3d." [Online]. Available: http://zach.in.tu-clausthal.de/teaching/cg_literatur/glsl_tutorial/

[7] "Opengl 4 tessellation." [Online]. Available: http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/

[8] G. Nunes, A. Valdetaro, A. Raposo, and B. Feijo, "Understanding shader model 5.0 with directx 11," in *Tutorial of the IX Brazilian Symposium on Computer Games and Digital Entertainment*, ser. SBGAMES '10, 2010.

[9] "Shader desinger." [Online]. Available: http://www.opengl.org/sdk/tools/ShaderDesigner/

[10] "Rendermonkey." [Online]. Available: http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx

[11] "Shader maker." [Online]. Available: http://cg.in.tu-clausthal.de/teaching/shader_maker/index.shtml

[12] F. Durand and B. Cutler, "6.837 computer graphics," vol. 13, pp. 18–28, 2003. [Online]. Available: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-837-computer-graphics-fall-2003/

[13] O. A. R. Board, D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1*, 6th ed. Addison-Wesley Professional, 2007.

[14] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, no. 6, pp. 311–317, Jun. 1975. [Online]. Available: http://doi.acm.org/10.1145/360825.360839

[15] H. Gouraud, "Continuous shading of curved surfaces," *IEEE Trans. Comput.*, vol. 20, no. 6, pp. 623–629, Jun. 1971. [Online]. Available: http://dx.doi.org/10.1109/T-C.1971.223313

[16] K. Perlin, "An image synthesizer," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 287–296, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/325165.325247

[17] "Vertex noise, nvidia sdk 9.52 code samples." [Online]. Available: http://developer.download.nvidia.com/SDK/9.5/Samples/samples.html#glsl_vnoise

[18] R. Toledo and B. Levy, "Extending the graphic pipeline with new gpu-accelerated primitives," in *International gOcad Meeting, Nancy, France*, 2004, also presented in Visgraf Seminar 2004, IMPA, Rio de Janeiro, Brazil. [Online]. Available: http://www.tecgraf.puc-rio.br/~rtoledo/publications

[19] M. Segal and K. Akeley, "The opengl(r) graphics system: A specification," 2010.

[20] O. Mallo, R. Peikert, C. Sigg, and F. Sadlo, "Illuminated lines revisited," in *IEEE Visualization'05*, 2005, pp. –1–1.

[21] M. Bailey, "Glsl geometry shaders." [Online]. Available: http://web.engr.oregonstate.edu/~mjb/cs519/Handouts/geometry_shaders.6pp.pdf

[22] D. T. Sato, H. Corbucci, and M. V. Bravo, "Coding dojo: An environment for learning and sharing agile practices," in *Proceedings of the Agile 2008*, ser. AGILE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 459–464. [Online]. Available: http://dx.doi.org/10.1109/Agile.2008.11

[23] T. Gomes and F. Vianna, "Shaderlabs: Desenvolvimento ágil de uma ide para opengl shaders," 2011. [Online]. Available: http://www.dcc.ufrj.br/~shaderlabs/files/monografia.pdf