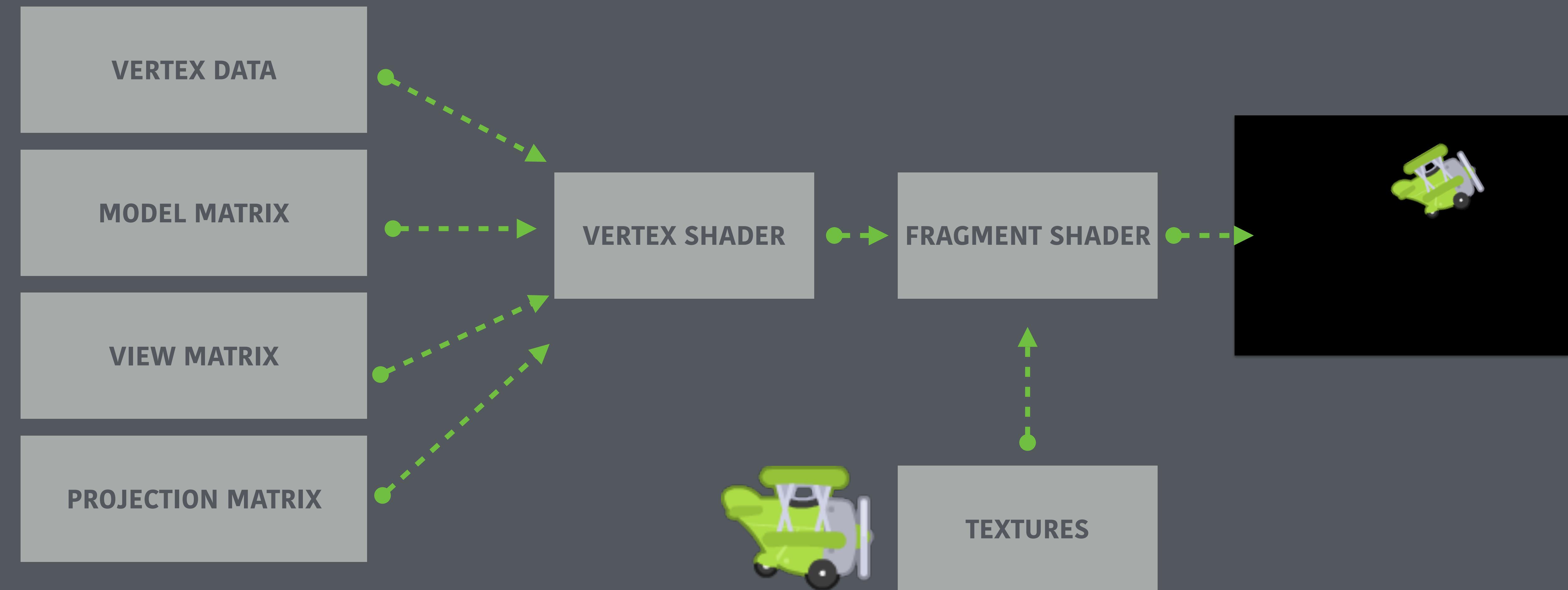


Shaders

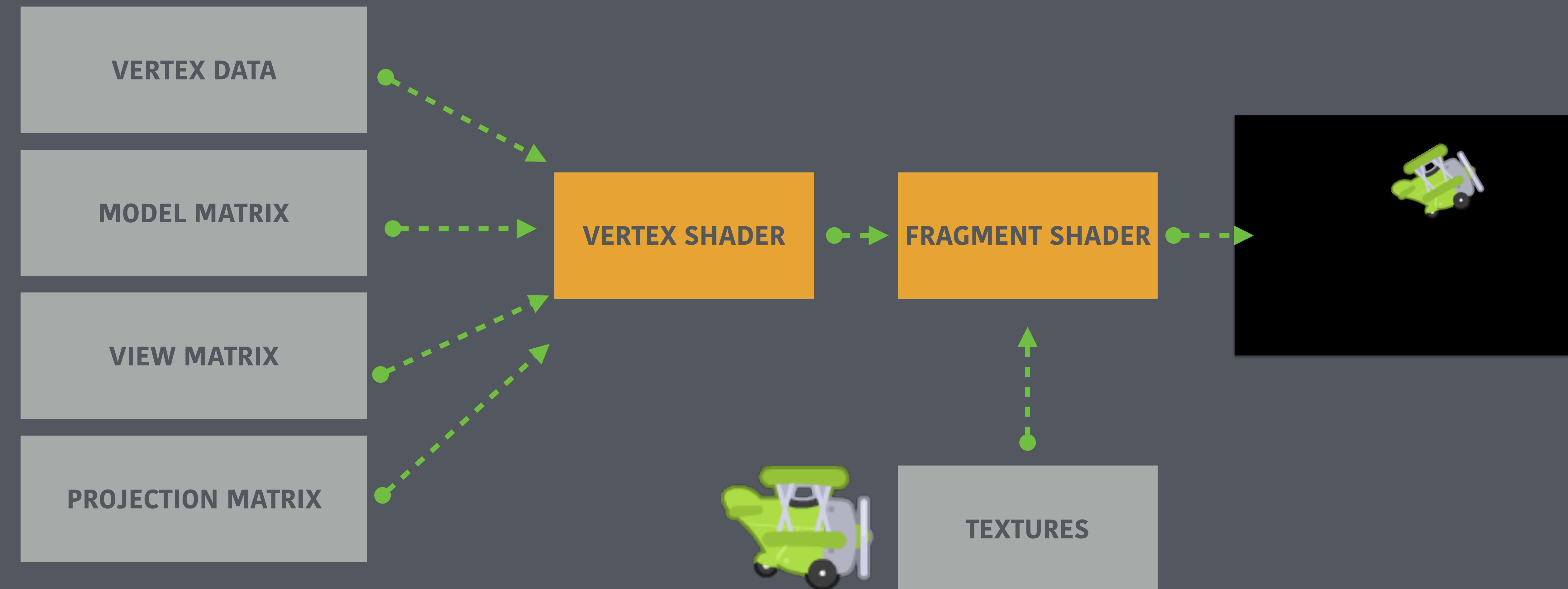
Part 1



The GPU pipeline



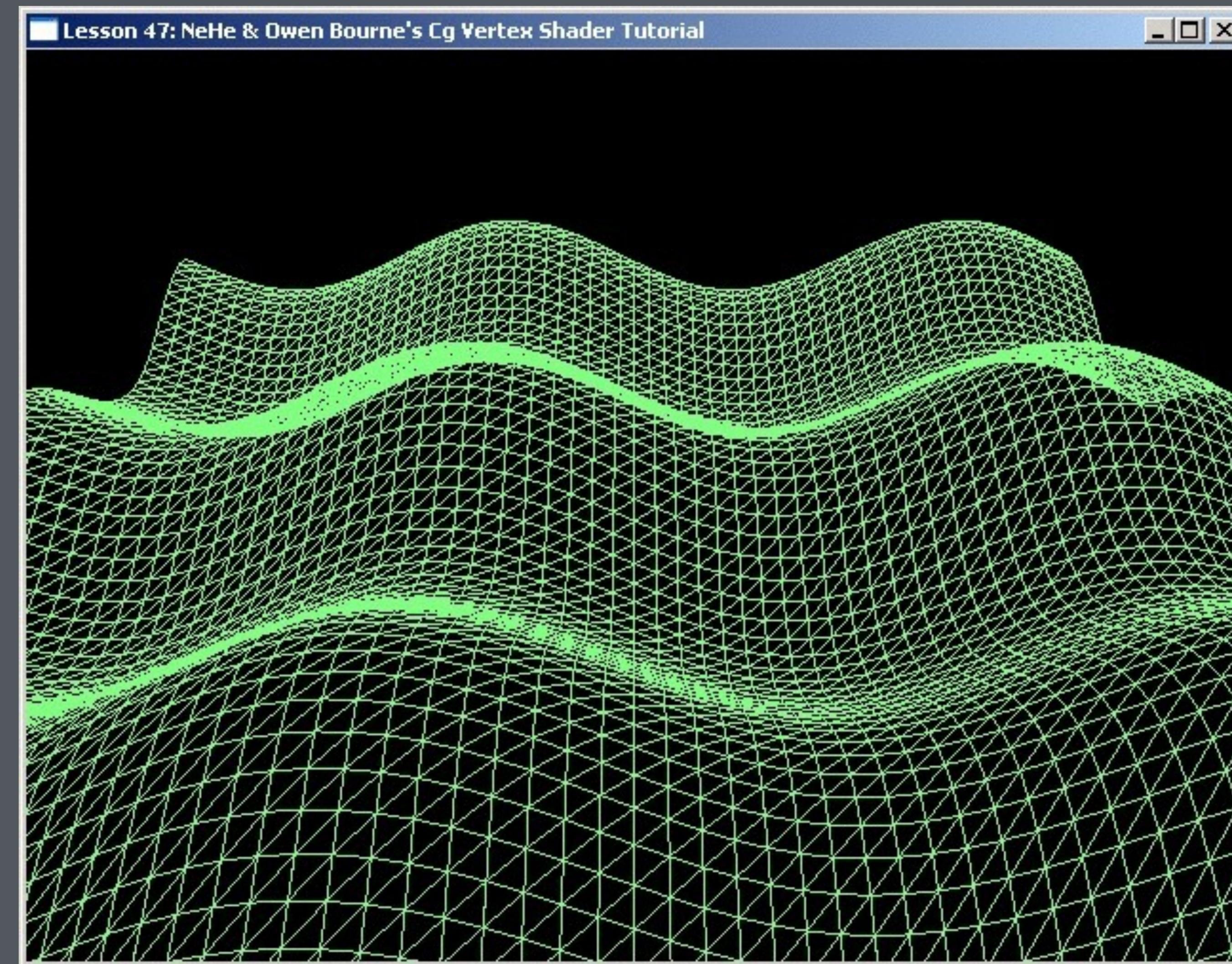
The GPU pipeline

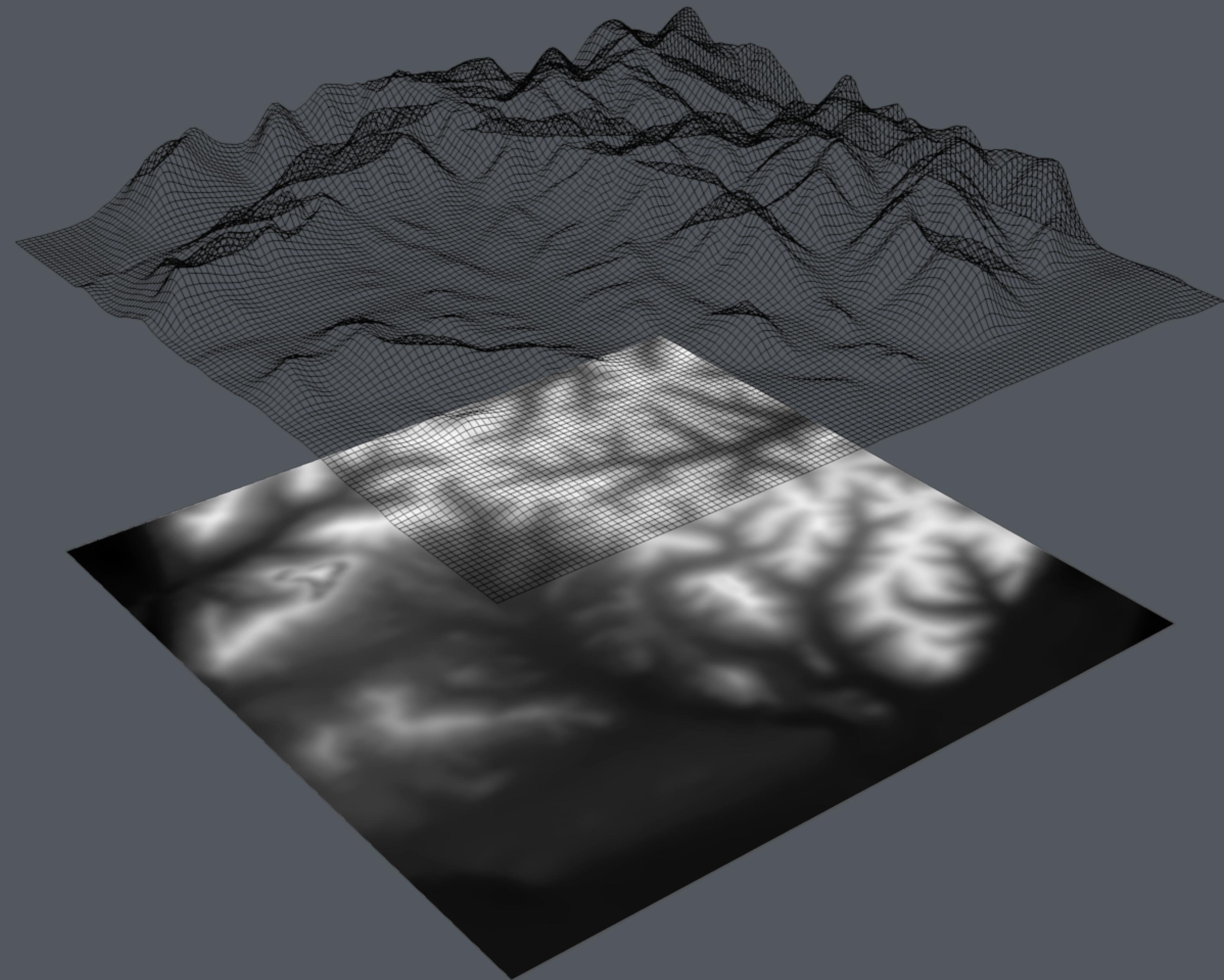


The vertex shader

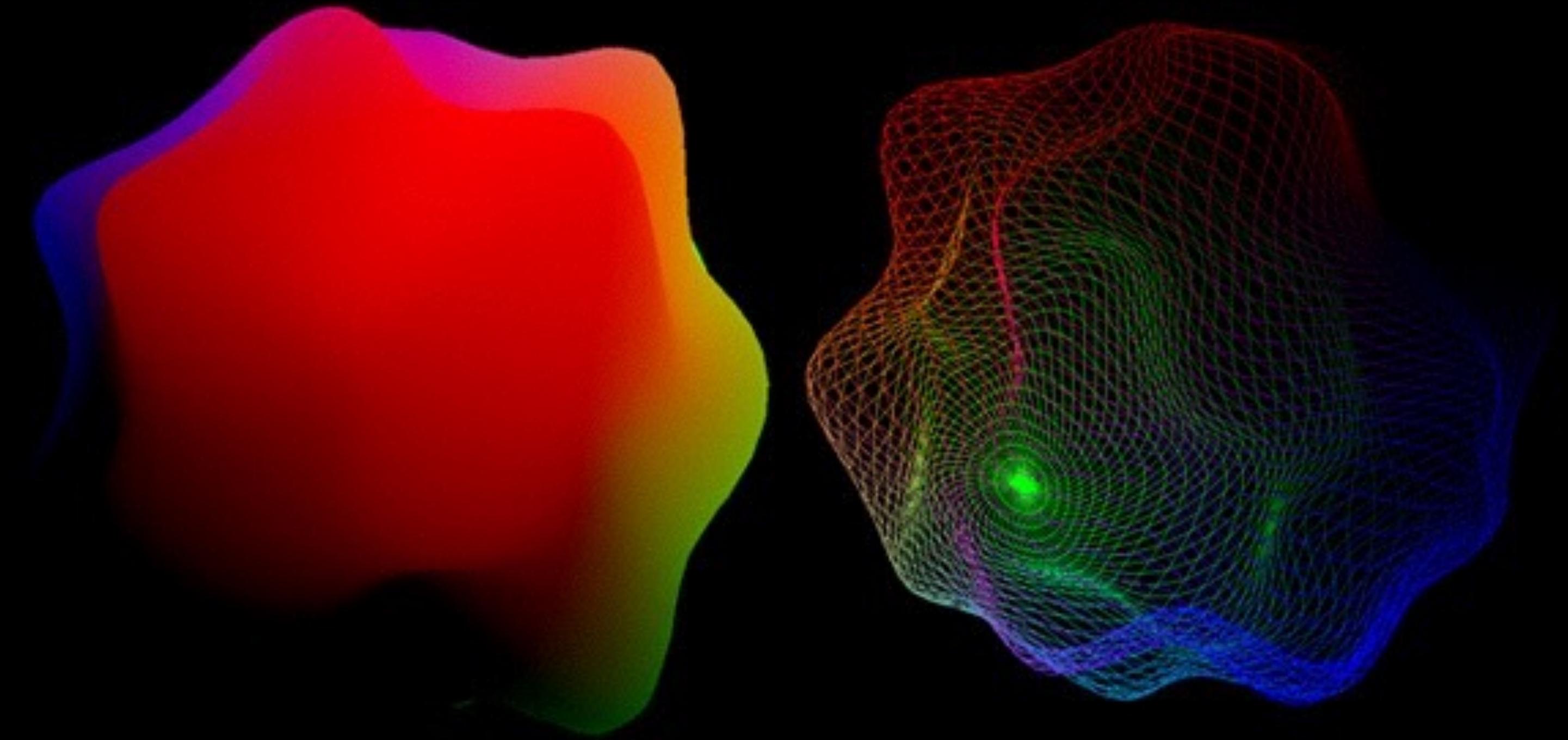
**A program that transforms the properties of
every vertex passed to the GPU
and passes data to the fragment shader.**

The **vertex** shader is where the **model**, **view** and
projection matrices are applied to the **vertex**.



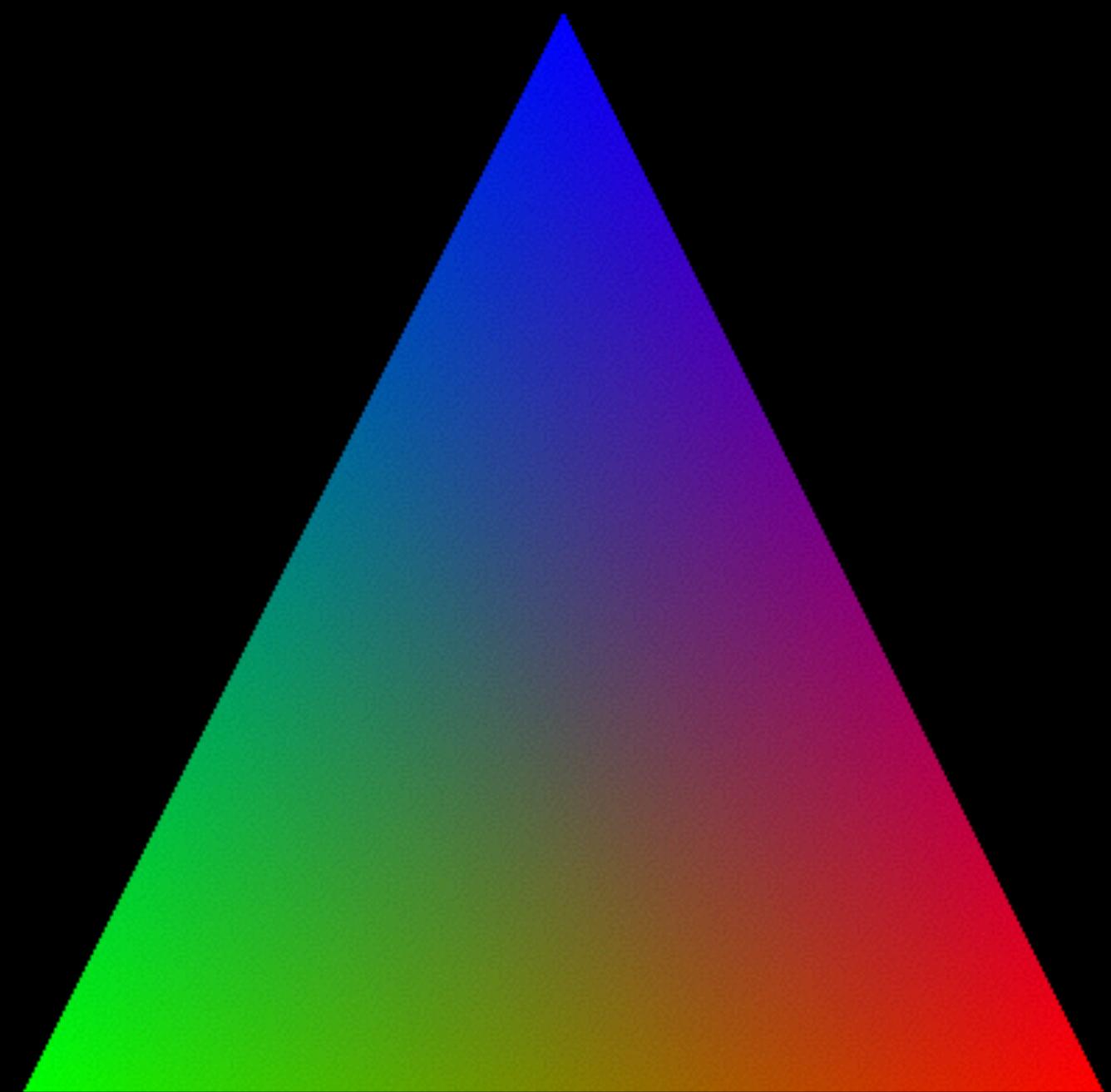


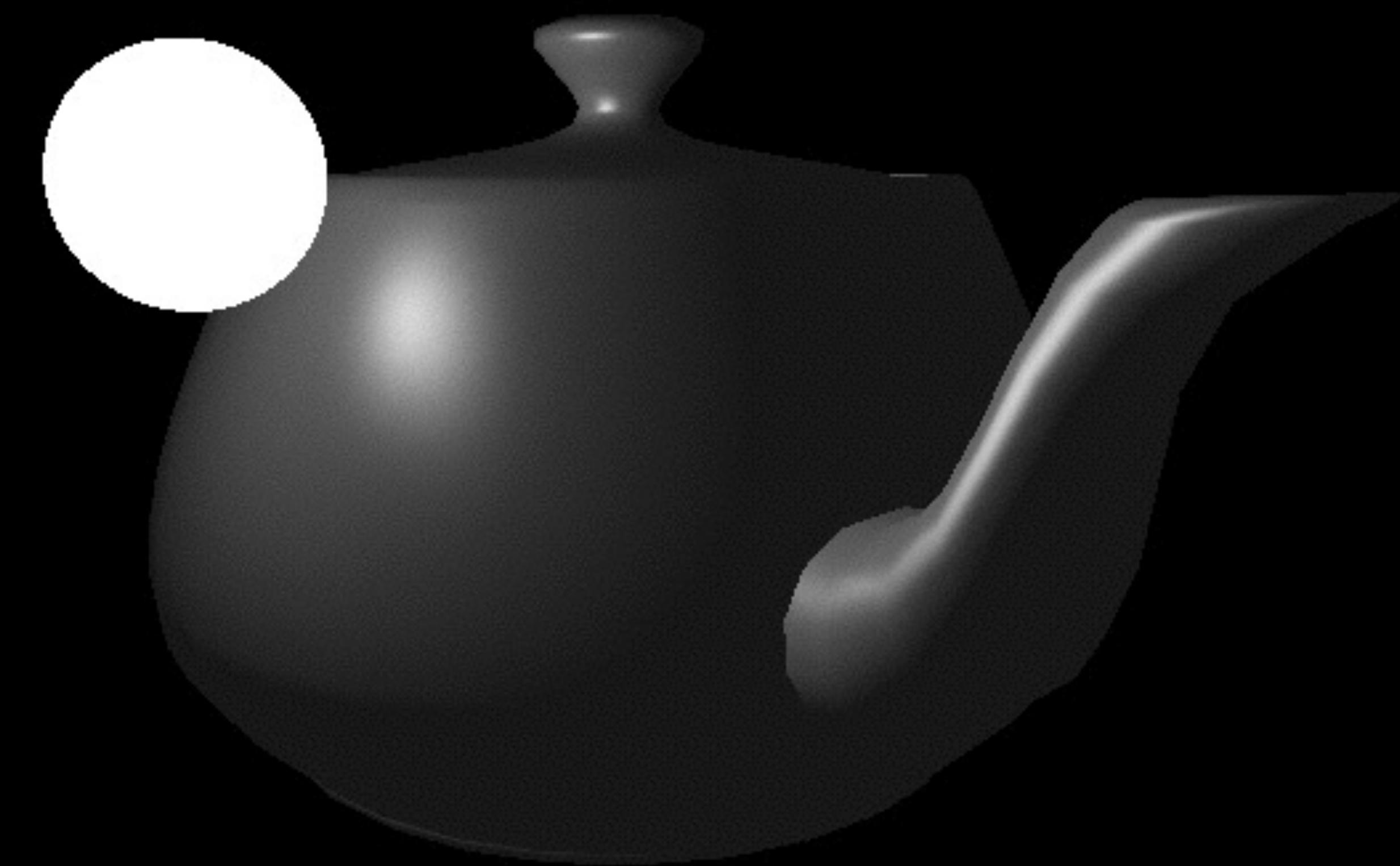


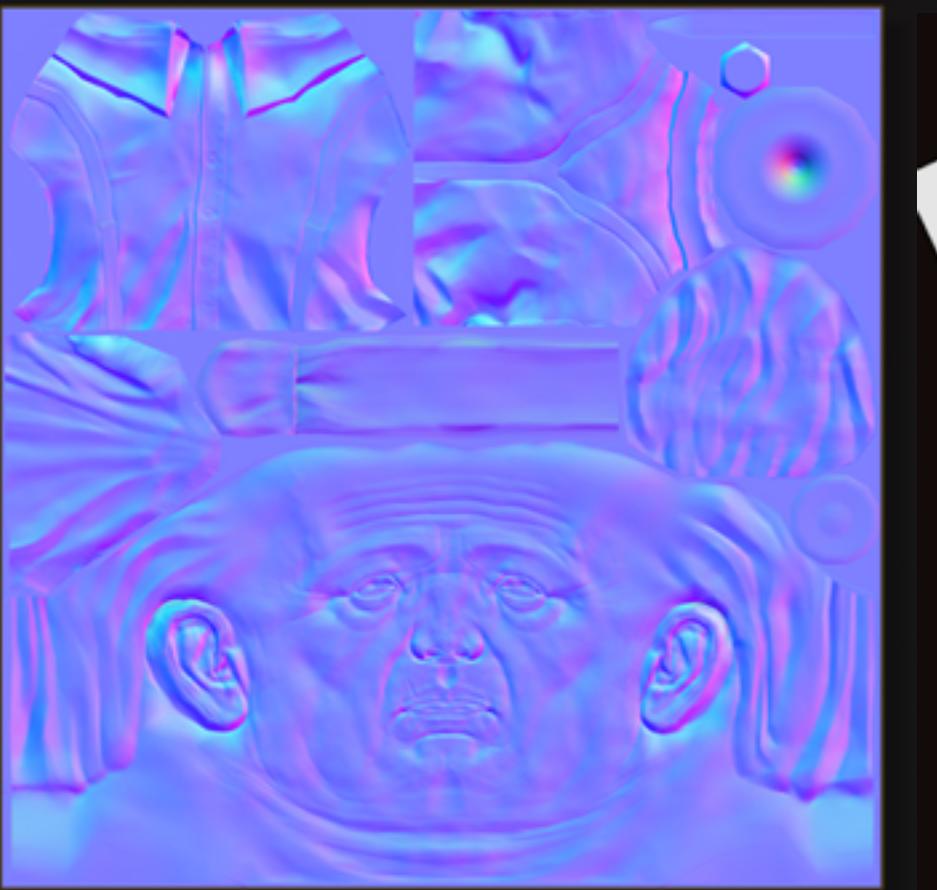


The fragment shader

A program that **returns the color of each pixel**
when geometry is rasterized on the GPU.







2K
CZECH
ART TEST
2048x2048 TEXTURES
2754 TRIS
WWW.ALEXANDERDELAGRANGE.COM





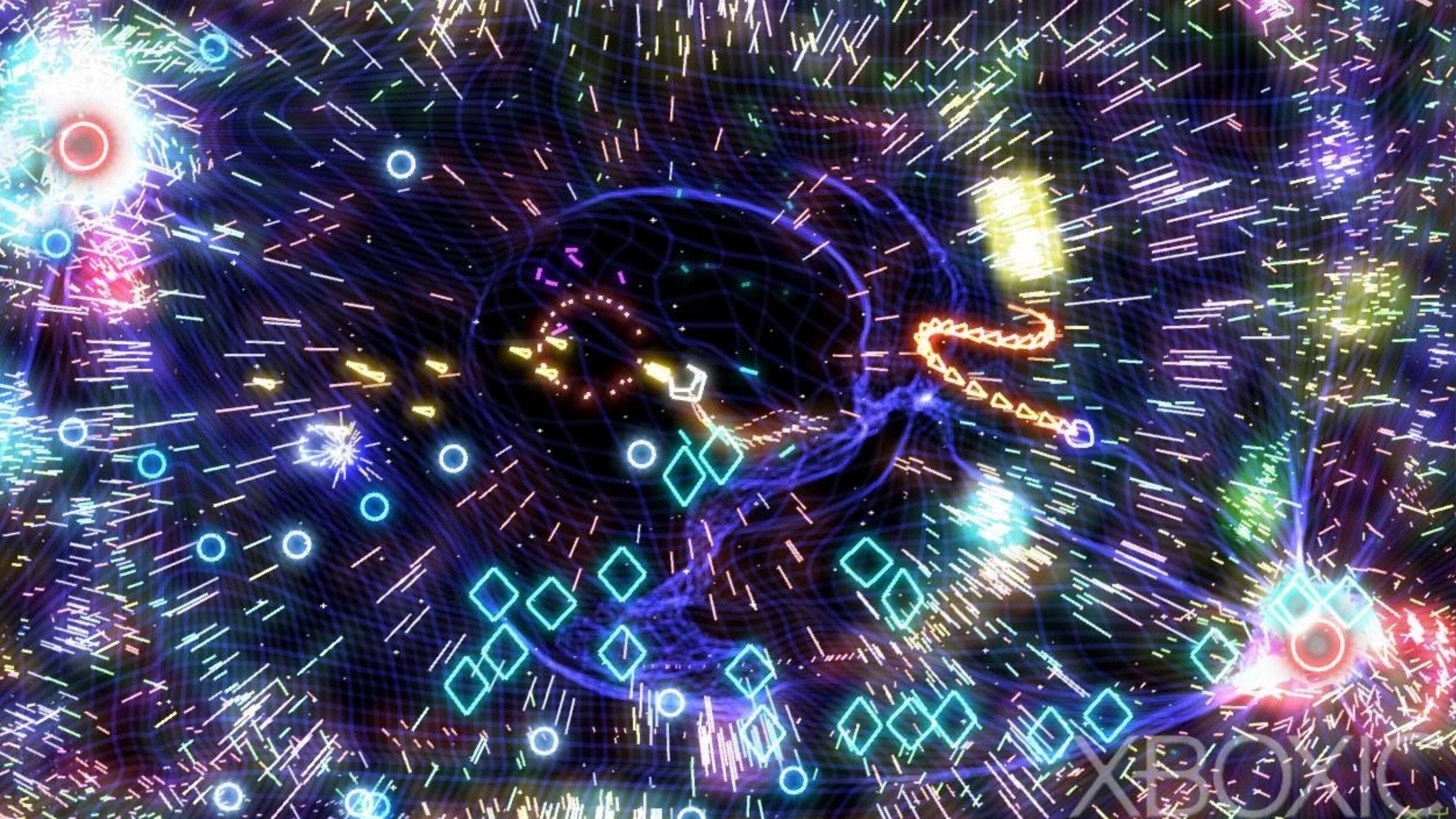




4 4 4

\$ 0





XBOX

The final **shader program** is a combination of
a **vertex** and a **fragment** shader.

GLSL

OpenGL Shading Language

Anatomy of a simple GLSL vertex shader.

```
attribute vec4 position;
attribute vec2 texCoord;

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

varying vec2 texCoordVar;

void main()
{
    vec4 p = viewMatrix * modelMatrix * position;
    texCoordVar = texCoord;
    gl_Position = projectionMatrix * p;
}
```

Anatomy of a simple GLSL vertex shader.

```
attribute vec4 position;  
attribute vec2 texCoord;
```

ATTRIBUTES

```
uniform mat4 modelMatrix;  
uniform mat4 viewMatrix;  
uniform mat4 projectionMatrix;
```

UNIFORMS

```
varying vec2 texCoordVar;
```

VARYING VARIABLES

```
void main()  
{  
    vec4 p = viewMatrix * modelMatrix * position;  
    texCoordVar = texCoord;  
    gl_Position = projectionMatrix * p;  
}
```

Anatomy of a simple **GLSL** fragment shader.

```
uniform sampler2D diffuse;  
  
varying vec2 texCoordVar;  
  
void main() {  
    gl_FragColor = texture2D(diffuse, texCoordVar);  
}
```

Anatomy of a simple GLSL fragment shader.

```
uniform sampler2D diffuse;  
  
varying vec2 texCoordVar;  
  
void main() {  
    gl_FragColor = texture2D(diffuse, texCoordVar);  
}
```

UNIFORMS

VARYING VARIABLES

Using shaders in OpenGL.

Loading and compiling vertex
and fragment shaders.

vertex_shader.glsl

```
attribute vec4 position;
attribute vec2 texCoord;

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

varying vec2 texCoordVar;

void main()
{
    vec4 p = viewMatrix * modelMatrix * position;
    texCoordVar = texCoord;
    gl_Position = projectionMatrix * p;
}
```

fragment_shader.glsl

```
uniform sampler2D diffuse;  
  
varying vec2 texCoordVar;  
  
void main() {  
    gl_FragColor = texture2D(diffuse, texCoordVar);  
}
```

Load our shader files into an **std::string**.

```
std::ifstream infile("vertex_shader.frag");
if(infile.fail()) {
    std::cout << "Error opening shader file" << std::endl;
}
std::stringstream buffer;
buffer << infile.rdbuf();
std::string vertexShader = buffer.str();
```

Creating and **compiling** a shader from string.

Use GL_VERTEX_SHADER for vertex shader.

```
GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);

const char *vertexShaderString = vertexShader.c_str();
int vertexShaderStringLength = vertexShader.size();

glShaderSource(vertexShaderID, 1, &vertexShaderString, &vertexShaderStringLength);
glCompileShader(vertexShaderID);
```

Use **GL_FRAGMENT_SHADER** for fragment shader.

```
GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

const char *fragmentShaderString = fragmentShader.c_str();
int fragmentShaderStringLength = fragmentShader.size();

glShaderSource(fragmentShaderID, 1, &fragmentShaderString, &fragmentShaderStringLength);
glCompileShader(fragmentShaderID);
```

Creating and linking the shader program from the fragment and vertex shaders.

```
GLuint exampleProgram = glCreateProgram();
glAttachShader(exampleProgram, vertexShaderID);
glAttachShader(exampleProgram, fragmentShaderID);
glLinkProgram(exampleProgram);
```

Get references to the **attribute** and
uniform locations of the shader
program so we can **pass data to it.**

```
attribute vec4 position;  
attribute vec2 texCoord;
```

ATTRIBUTES

```
uniform mat4 modelMatrix;  
uniform mat4 viewMatrix;  
uniform mat4 projectionMatrix;
```

UNIFORMS

Uniforms

```
GLint projectionMatrixUniform = glGetUniformLocation(exampleProgram, "projectionMatrix");
GLint modelMatrixUniform = glGetUniformLocation(exampleProgram, "modelMatrix");
GLint viewMatrixUniform = glGetUniformLocation(exampleProgram, "viewMatrix");
```

Attributes

```
GLuint positionAttribute = glGetAttribLocation(exampleProgram, "position");
GLuint texCoordAttribute = glGetAttribLocation(exampleProgram, "texCoord");
```

Drawing using our shader program.

Tell OpenGL to use our program.

```
glUseProgram(exampleProgram);
```

Bind our model, view and projection matrices.

```
glUniformMatrix4fv(projectionMatrixUniform, 1, GL_FALSE, projectionMatrix.ml);
glUniformMatrix4fv(modelMatrixUniform, 1, GL_FALSE, modelMatrix.ml);
glUniformMatrix4fv(viewMatrixUniform, 1, GL_FALSE, viewMatrix.ml);
```

Bind shader attributes.

```
float vertices[] = {-0.5, 0.5, -0.5, -0.5, 0.5, -0.5, -0.5, 0.5, 0.5, -0.5, 0.5, 0.5};  
glVertexAttribPointer(positionAttribute, 2, GL_FLOAT, false, 0, vertices);  
 glEnableVertexAttribArray(positionAttribute);  
  
float texCoords[] = {0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0};  
glVertexAttribPointer(texCoordAttribute, 2, GL_FLOAT, false, 0, texCoords);  
 glEnableVertexAttribArray(texCoordAttribute);  
  
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Catching GLSL errors when compiling
shaders.

After calling `glCompileShader` you can check if it failed to compile and get a readable error message.

```
GLint compileSuccess;
glGetShaderiv(vertexShaderID, GL_COMPILE_STATUS, &compileSuccess);
if (compileSuccess == GL_FALSE) {

    GLchar messages[256];
    glGetShaderInfoLog(shaderID, sizeof(messages), 0, &messages[0]);
    std::cout << messages << std::endl;
}
```

Intro to basic GLSL

Basic Types

void	no function return value or empty parameter list
bool	Boolean
int	signed integer
float	floating scalar
vec2, vec3, vec4	n-component floating point vector
bvec2, bvec3, bvec4	Boolean vector
ivec2, ivec3, ivec4	signed integer vector
mat2, mat3, mat4	2x2, 3x3, 4x4 float matrix
sampler2D	access a 2D texture
samplerCube	access cube mapped texture

Vertex shader must set **gl_Position** (which is a homogenized **vec4** coordinate) to set the final position of the vertex being drawn.

If it is passing any **varying** data to the pixel shader, it must set those as well.

```
attribute vec4 position;
attribute vec2 texCoord;

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

varying vec2 texCoordVar;

void main()
{
    vec4 p = viewMatrix * modelMatrix * position;
    texCoordVar = texCoord;
    gl_Position = projectionMatrix * p;
}
```

Fragment shader must set **gl_FragColor** (which is a **vec4 RGBA color value**) to set the final color of the pixel being rendered.

The **texture2D** function is built in and takes a **sampler2D texture** and a **vec2 UV coordinate** and returns a **vec4 RGBA color** from that texture at that coordinate.

```
uniform sampler2D diffuse;  
  
varying vec2 texCoordVar;  
  
void main() {  
    gl_FragColor = texture2D(diffuse, texCoordVar);  
}
```

Simple shader examples.

Inverting texture color.

```
uniform sampler2D texture;  
  
varying vec2 texCoordVar;  
  
void main()  
{  
    vec4 finalColor = 1.0 - texture2D( texture, texCoordVar);  
    finalColor.a = texture2D( texture, texCoordVar).a;  
    gl_FragColor = finalColor;  
}
```

Making a texture black and white.

```
uniform sampler2D texture;  
  
varying vec2 texCoordVar;  
  
void main()  
{  
    vec4 texColor = texture2D( texture, texCoordVar);  
    vec4 finalColor = vec4((texColor.r + texColor.g + texColor.b)/3.0);  
    finalColor.a = texture2D( texture, texCoordVar).a;  
    gl_FragColor = finalColor;  
}
```

Functions in GLSL

Built-In Functions

Angle & Trigonometry Functions [8.1]

Component-wise operation. Parameters specified as *angle* are assumed to be in units of radians. T is float, vec2, vec3, vec4.

T radians(T degrees)	degrees to radians
T degrees(T radians)	radians to degrees
T sin(T angle)	sine
T cos(T angle)	cosine
T tan(T angle)	tangent
T asin(T x)	arc sine
T acos(T x)	arc cosine
T atan(T y, T x)	arc tangent
T atan(T y_over_x)	

Exponential Functions [8.2]

Component-wise operation. T is float, vec2, vec3, vec4.

T pow(T x, T y)	x^y
T exp(T x)	e^x
T log(T x)	\ln
T exp2(T x)	2^x
T log2(T x)	\log_2
T sqrt(T x)	square root
T inversesqrt(T x)	inverse square root

Common Functions [8.3]

Component-wise operation. T is float, vec2, vec3, vec4.

T abs(T x)	absolute value
T sign(T x)	returns -1.0, 0.0, or 1.0
T floor(T x)	nearest integer $\leq x$
T ceil(T x)	nearest integer $\geq x$
T fract(T x)	$x - \text{floor}(x)$
T mod(T x, T y) T mod(T x, float y)	modulus
T min(T x, T y) T min(T x, float y)	minimum value
T max(T x, T y) T max(T x, float y)	maximum value
T clamp(T x, T minVal, T maxVal) T clamp(T x, float minVal, float maxVal)	$\min(\max(x, \text{minVal}), \text{maxVal})$
T mix(T x, T y, T a) T mix(T x, T y, float a)	linear blend of x and y
T step(T edge, T x) T step(float edge, T x)	0.0 if $x < \text{edge}$, else 1.0
T smoothstep(T edge0, T edge1, T x) T smoothstep(float edge0, float edge1, T x)	clip and smooth

Geometric Functions [8.4]

These functions operate on vectors as vectors, not component-wise. T is float, vec2, vec3, vec4.

T length(T x)	length of vector
T distance(T p0, T p1)	distance between points
T dot(T x, T y)	dot product
T cross(vec3 x, vec3 y)	cross product
T normalize(T x)	normalize vector to length 1
T faceforward(T N, T I, T Nref)	returns N if $\dot{Nref, I} < 0$, else $-N$
T reflect(T I, T N)	reflection direction $I - 2 * \dot{N, I} * N$
T refract(T I, T N, float eta)	refraction vector

Matrix Functions [8.5]

Type mat is any matrix type.

mat matrixCompMult(mat x, mat y)	multiply x by y component-wise
---	--------------------------------

Vector Relational Functions [8.6]

Compare x and y component-wise. Sizes of input and return vectors for a particular call must match. Type bvec is bvecn; vec is vecn; ivec is ivec n (where n is 2, 3, or 4). T is the union of vec and ivec.

bvec lessThan(T x, T y)	$x < y$
bvec lessThanEqual(T x, T y)	$x \leq y$
bvec greaterThan(T x, T y)	$x > y$
bvec greaterThanEqual(T x, T y)	$x \geq y$
bvec equal(T x, T y)	$x == y$
bvec equal(bvec x, bvec y)	
bvec notEqual(T x, T y)	$x != y$
bvec notEqual(bvec x, bvec y)	
bool any(bvec x)	true if any component of x is true
bool all(bvec x)	true if all components of x are true
bvec not(bvec x)	logical complement of x

Texture Lookup Functions [8.7]

Available only in vertex shaders.

vec4 texture2DLod(sampler2D sampler, vec2 coord, float lod)
vec4 texture2DProjLod(sampler2D sampler, vec3 coord, float lod)
vec4 texture2DProjLod(sampler2D sampler, vec4 coord, float lod)
vec4 textureCubeLod(samplerCube sampler, vec3 coord, float lod)

Available only in fragment shaders.

vec4 texture2D(sampler2D sampler, vec2 coord, float bias)
vec4 texture2DProj(sampler2D sampler, vec3 coord, float bias)
vec4 texture2DProj(sampler2D sampler, vec4 coord, float bias)
vec4 textureCube(samplerCube sampler, vec3 coord, float bias)

Available in vertex and fragment shaders.

vec4 texture2D(sampler2D sampler, vec2 coord)
vec4 texture2DProj(sampler2D sampler, vec3 coord)
vec4 texture2DProj(sampler2D sampler, vec4 coord)
vec4 textureCube(samplerCube sampler, vec3 coord)

Saturation function example.

```
uniform sampler2D texture;
varying vec2 texCoordVar;

vec3 saturation_func(vec3 rgb, float adjustment)
{
    const vec3 W = vec3(0.2125, 0.7154, 0.0721);
    vec3 intensity = vec3(dot(rgb, W));
    return mix(intensity, rgb, adjustment);
}

void main()
{
    vec4 finalColor;
    finalColor.rgb = saturation_func(texture2D( texture, texCoordVar).rgb, 2.0);
    finalColor.a = texture2D( texture, texCoordVar).a;
    gl_FragColor = finalColor;
}
```

Passing variables as uniforms.

Example: passing the saturation value from our C++ code.

```
uniform sampler2D texture;
uniform float saturationAmount;
varying vec2 texCoordVar;

vec3 saturation_func(vec3 rgb, float adjustment)
{
    const vec3 W = vec3(0.2125, 0.7154, 0.0721);
    vec3 intensity = vec3(dot(rgb, W));
    return mix(intensity, rgb, adjustment);
}

void main()
{
    vec4 finalColor;
    finalColor.xyz = saturation_func(texture2D( texture, texCoordVar).xyz, saturationAmount);
    finalColor.a = texture2D( texture, texCoordVar).a;
    gl_FragColor = finalColor;
}
```

Get the saturation amount uniform location.

```
GLint saturationAmountUniform = glGetUniformLocation(exampleProgram, "saturationAmount");
```

Before rendering, bind a value to it.

```
glUniform1f(saturationAmountUniform, 2.0f);
```

Some GLSL types and their corresponding C++ uniform binding functions.

float - **glUniform1f**(location, value);

vec2 - **glUniform2f**(location, value1, value2);

vec3 - **glUniform3f**(location, value1, value2, value3);

vec4 - **glUniform4f**(location, value1, value2, value3, value4);

Example: scrolling a texture.

```
uniform sampler2D texture;
uniform vec2 scroll;
varying vec2 texCoordVar;

void main()
{
    gl_FragColor = texture2D( texture, texCoordVar + scroll);
}
```

Get the scroll amount **uniform location**.

```
GLint scrollUniform = glGetUniformLocation(exampleProgram, "scroll");
```

Before rendering, bind a value to it.

```
glUniform2f(scrollUniform, ticks, 0.0f);
```

Final project requirements.

- Must have a title screen and proper states for game over, etc.
- Must have a way to quit the game.
- Must have music and sound effects.
- Must have at least 3 different levels or be procedurally generated.
- Must be either local multiplayer or have AI (or both!).
- Must have at least some animation or particle effects.

Bonus points for...

- Getting it running on your phone.
- Having 3D elements.
- Having shader effects.

(we haven't covered any of this yet!)