

OpenGL Shading Language (GLSL)

OpenGL Shading Language (GLSL) is a high-level shading language with a syntax based on the C programming language. It runs natively on OpenGL but can also run on Vulkan by using Spir-V

- [Core C Programming for GLSL](#)
- [Fragment Shaders](#)
- [Converting ShaderToy to GLSL](#)

Core C Programming for GLSL

Writing GLSL (OpenGL Shading Language) code does not require extensive knowledge of C programming, but familiarity with some core C concepts can be very helpful. GLSL shares many similarities with C, as it is a C-like language, but there are important differences as well.

Here's a breakdown of how much C programming you need to know to effectively write GLSL shaders:

Core C Concepts Useful for GLSL:

1. ****Data Types and Variables****:

- GLSL has many data types similar to C, such as `int`, `float`, and `bool`, but it also includes types specific to graphics, such as `vec2`, `vec3`, `vec4` (vectors), and `mat4` (matrices).
- Knowing how to declare variables and use basic data types from C is useful.

```
```c
```

```
// C:
```

```
int x = 5;
```

```
float y = 3.14;
```

```
// GLSL:
```

```
int x = 5;
```

```
float y = 3.14;
```

```
vec3 color = vec3(1.0, 0.5, 0.0); // A 3D vector for color
```

```
```
```

2. ****Control Structures****:

- C-like control structures, such as `if`, `else`, `for`, and `while`, are available in GLSL.
- Basic knowledge of how to write loops and conditional statements in C will help in GLSL.

```
```c
```

```
// C:
```

```
for (int i = 0; i < 10; i++) {
```

```
 if (i % 2 == 0) {
```

```
 // Do something
```

```
 }
```

```
}
```

```
// GLSL:
```

```
for (int i = 0; i < 10; i++) {
```

```
 if (i % 2 == 0) {
```

```

 // Do something, like color a pixel
}
}
...

```

### 3. **Functions**:

- GLSL, like C, allows you to write functions that take parameters, return values, and encapsulate logic.

- Knowledge of defining and calling functions in C will help in GLSL.

```

...C
// C:
int add(int a, int b) {
 return a + b;
}

// GLSL:
float add(float a, float b) {
 return a + b;
}
...

```

### 4. **Arrays**:

- Similar to C, GLSL supports arrays. Knowing how to declare and use arrays in C translates directly into GLSL.

```

...C
// C:
int arr[3] = {1, 2, 3};

// GLSL:
float arr[3] = float[3](1.0, 2.0, 3.0);
...

```

### 5. **Mathematical Operators**:

- Like C, GLSL uses standard arithmetic (`+`, `-`, `*`, `/`) and relational operators (`<`, `>`, `==`). Knowing basic arithmetic and logic operators from C will help in writing shader code.

- In addition, GLSL has built-in functions like `sin()`, `cos()`, `dot()`, `cross()`, etc., for vector and matrix math, which are unique to graphics programming.

### 6. **Type Casting**:

- Understanding type casting in C helps, although GLSL type casting is simpler. For instance, casting between different numeric types or vector types is common in GLSL.

```

...C
// C:
float f = (float)5;

// GLSL:

```

```
float f = float(5); // Casts int to float
...
```

### ### GLSL-Specific Concepts (Beyond C):

While C programming fundamentals are useful, there are several key areas where GLSL differs from C, especially since it is designed specifically for graphics.

#### 1. **Vectors and Matrices**:

- GLSL has built-in types for vectors (`vec2`, `vec3`, `vec4`) and matrices (`mat3`, `mat4`) to handle graphics-specific tasks like transformations, lighting, and colors. You won't find these in C, so you'll need to learn how to work with these types in GLSL.
- GLSL allows vector arithmetic and component-wise operations, which are not natively supported in C.

```
...glsl
vec3 position = vec3(1.0, 2.0, 3.0);
vec3 direction = normalize(position); // Normalize vector
mat4 transform = mat4(1.0); // 4x4 identity matrix
...
```

#### 2. **Shader Stages**:

- Unlike C, GLSL is executed in the context of a **graphics pipeline**, with specific shader stages like **vertex shaders**, **fragment shaders**, **geometry shaders**, etc. Each stage has a specific role (e.g., vertex shaders handle vertex transformations, fragment shaders handle per-pixel color calculations).
- Knowing how data flows through the pipeline and between shaders is essential for writing GLSL, but this isn't part of C programming.

#### 3. **Uniforms and Inputs/Outputs**:

- In GLSL, shaders use **uniforms** (global variables passed from the CPU to the GPU) and input/output variables to communicate between different shader stages. These don't exist in C, but learning to use them is crucial for GLSL programming.

```
...glsl
uniform mat4 modelViewMatrix; // Passed from the application
in vec3 vertexPosition; // Input from vertex data
out vec4 fragColor; // Output to fragment shader
...
```

#### 4. **Built-in Functions**:

- GLSL has a large set of built-in functions for graphics-specific operations, like texture sampling (`texture()`), lighting calculations, and geometric functions (`dot()`, `cross()`, etc.). These are not part of C, and you'll need to learn how to use them effectively.

#### 5. **No Pointers**:

- Unlike C, GLSL does not support **pointers** or **manual memory management**. This simplifies the language somewhat compared to C, but it also means you can't use some of the lower-level features that C provides.

### ### Summary: How Much C Do You Need?

- **Basic Knowledge** of C is sufficient for learning GLSL. Understanding variables, control structures (loops, conditionals), functions, and arrays from C will help you write shader code.
- **Advanced C Concepts** like pointers, manual memory management, and structs are not needed in GLSL.
- However, you'll need to learn **GLSL-specific features** like vectors, matrices, built-in functions, and the graphics pipeline, which are beyond the scope of C.

In short, **if you are comfortable with basic C programming**, you'll find it relatively easy to pick up GLSL with some additional learning of its graphics-specific features.

Let me know if you need further clarification or examples!

# Fragment Shaders

Fragment shader code is executed entirely on the **GPU** (Graphics Processing Unit).

1. **Parallelism:** The GPU is designed to handle many operations at once, and since fragment shaders need to process every pixel on the screen, the GPU's architecture allows these operations to be run in parallel. Each fragment (which corresponds to a pixel or a portion of a pixel) is processed independently by the fragment shader
2. **Efficiency:** Fragment shaders handle tasks like lighting, color blending, texturing, and other pixel-level effects. These operations are computationally intensive but relatively simple when done for each pixel. The CPU would be far less efficient at handling millions of pixels in real-time.

## Process Overview:

When rendering a scene:

1. **Vertices** are processed in the vertex shader.
2. **Rasterization** occurs, converting geometric shapes (like triangles) into fragments (potential pixels).
3. The **fragment shader** then runs on each fragment to compute its color, texture, and other attributes, often involving texture sampling and lighting calculations.
4. After the fragment shader, the final color values are written to the framebuffer, creating the final image.

## Example:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord) {
 // Compute UV coordinates
 vec2 uv = fragCoord.xy / iResolution.xy;

 // Set the output color
 fragColor = vec4(uv, 0.0, 1.0); // Sets a color based on UV
}
```

## Color Space

In GLSL (and other graphics APIs), colors are typically represented using RGB (Red, Green, Blue) values. These values can exist in two spaces: the **original space** (or "integer space") and the **normalized space** (or "floating-point space"). Here's how these two spaces differ:

## 1. Integer Space

- **Range:** Each color component (R, G, B) is usually represented as an integer between 0 and 255.
- **Representation:** The RGB values are discrete integers, where 0 represents no intensity (black) and 255 represents full intensity (maximum brightness) for each color component.
  - **Black:** (0, 0, 0) (all components are at their minimum)
  - **White:** (255, 255, 255) (all components are at their maximum)
  - **Red:** (255, 0, 0) (only the red component is at maximum)
  - **Green:** (0, 255, 0) (only the green component is at maximum)
  - **Blue:** (0, 0, 255) (only the blue component is at maximum)

### Example:

- A pixel with the color (128, 64, 255) in integer space would have:
  - **Red:** 128 (half brightness)
  - **Green:** 64 (quarter brightness)
  - **Blue:** 255 (full brightness)

## 2. Normalized Space (Floating-Point Space)

- **Range:** Each color component (R, G, B) is represented as a floating-point number between 0.0 and 1.0.
- **Representation:** In this space, 0.0 represents no intensity (black), and 1.0 represents full intensity (maximum brightness). Each integer value from the original space is divided by 255 to convert it into normalized space.
  - **Black:** (0.0, 0.0, 0.0) (all components are at their minimum)
  - **White:** (1.0, 1.0, 1.0) (all components are at their maximum)
  - **Red:** (1.0, 0.0, 0.0) (only the red component is at maximum)
  - **Green:** (0.0, 1.0, 0.0) (only the green component is at maximum)
  - **Blue:** (0.0, 0.0, 1.0) (only the blue component is at maximum)

### Conversion Formula:

To convert from the original space to normalized space:

$$\text{normalized value} = \frac{\text{original value}}{255}$$
$$\text{normalized value} = \text{original value} / 255$$

For example:

- A pixel with the color (128, 64, 255) in integer space would be represented as:
    - **Red:**  $128/255 \approx 0.502$
    - **Green:**  $64/255 \approx 0.251$
    - **Blue:**  $255/255 = 1.0$
- So, the color in normalized space would be approximately (0.502, 0.251, 1.0).

## Relationship Between Spaces:

- The conversion between the two spaces is linear. Integer values in the range [0, 255] are mapped to floating-point values in the range [0.0, 1.0].
- Integer space is typically used for storing pixel data in images and textures, while normalized space is used when performing operations in shaders.

## Example in GLSL:

In GLSL, colors are usually represented in **normalized space**. For example:

glsl

Copy code

```
vec4 color = vec4(1.0, 0.0, 0.0, 1.0); // Red in normalized space
```

Here, the red channel is set to 1.0, which corresponds to 255 in integer space, and the green and blue channels are 0.0, corresponding to 0 in integer space.

## Practical Use in Shaders:

When working with textures or colors in shaders:

- Textures sampled using the texture() function usually return colors in **normalized space**. For instance, a texture value might be returned as vec4(0.5, 0.3, 0.8, 1.0), where each component is in the [0.0, 1.0] range.
- If you want to manipulate colors in **integer space**, you would need to scale them up by 255 when necessary.

## Summary:

- **Original Space:** RGB values range from 0 to 255.
- **Normalized Space:** RGB values range from 0.0 to 1.0.
- Conversion between the two spaces involves dividing or multiplying by 255.

Would you like more details on any specific use of color spaces in shaders?



# Common GLSL Variables and Functions:

These functions could be specific to Shadertoy or OpenGL ES

- `fragCoord.xy`: Current pixel coordinates (in pixels).
- `iResolution.xy`: Screen resolution, used for normalization.
- `iTime`: Time since shader started, useful for animation.
- `iMouse`: Mouse position, used for interaction.
- `iChannel0`, `iChannel1`, **etc.**: Texture samplers for loading images or videos.
- `gl_FragCoord`: Built-in pixel coordinates in GLSL.
- `dot`, `cross`, `normalize`: Vector math functions for lighting and geometric operations.
- **Transformations**: Rotation, scaling, and translation to manipulate coordinates.
- `mod`, `fract`: Useful for creating repeating or tiled patterns.

## Getting Started

When you're working in shaders, the pixel coordinates from `fragCoord.xy` are typically in pixel units (e.g., ranging from `(0,0)` to `(1920,1080)` for an HD screen). If you want to make your shader work consistently across different resolutions, you need to normalize these coordinates.

By dividing `fragCoord.xy` by `iResolution.xy`, you map the pixel coordinates to a normalized space, which usually ranges from `[0, 1]`.

INCLUDE CODE HERE

### 1. `fragCoord.xy`:

- **Definition:** `fragCoord.xy` is a built-in variable in GLSL that represents the **coordinates of the current fragment (or pixel)** on the screen.
- **Values:** It provides the (x, y) pixel coordinates of the fragment being processed, starting from the bottom-left corner (which is `(0, 0)` in most shader coordinate systems) and going up to the width and height of the viewport (i.e., screen resolution).
- **Purpose:** It allows you to know the exact location of the pixel you're working with. For instance, if your window is 1920x1080, `fragCoord.xy` will range from `(0,0)` (bottom-left corner) to `(1920,1080)` (top-right corner).

### 2. `iResolution.xy`:

- **Definition:** `iResolution.xy` is a uniform variable specific to ShaderToy that represents the **dimensions of the screen (viewport)** in pixels, i.e., the resolution of the output

window.

- **Values:** `iResolution.x` is the width of the screen in pixels, and `iResolution.y` is the height of the screen in pixels.
- **Purpose:** It provides the resolution of the screen so that you can scale or normalize coordinates based on the size of the output window. This is important to make sure your shader adapts to different screen resolutions.

The `length` function in GLSL is a useful tool for calculating the magnitude (or Euclidean distance) of a vector. It's often used in shaders to compute distances between points or to create effects like radial gradients, circular shapes, and more.

## Definition:

- **`length(x)`:** Returns the Euclidean length (or magnitude) of the vector `x`. It works for both 2D, 3D, and 4D vectors.

# Converting ShaderToy to GLSL

ShaderToy is a platform for creating and sharing shaders in GLSL, but it includes some unique functions and features that are specific to its environment. If you want to port ShaderToy shaders to plain GLSL or another environment, you'll need to understand these specific functions and how to replace or adapt them.

Here are some common ShaderToy-specific functions and their equivalents or alternatives in standard GLSL:

1. **iGlobalTime**: This variable represents the time elapsed since the shader started running.

- **In GLSL**: You'll need to pass the elapsed time as a uniform variable. For example:

glsl

Copy code

```
uniform float uTime; // Use this uniform in your shader
```

2. **iResolution**: This variable holds the resolution of the output image.

- **In GLSL**: You can pass the resolution as a uniform variable:

glsl

Copy code

```
uniform vec2 uResolution; // Use this uniform in your shader
```

3. **iMouse**: This variable contains the position and state of the mouse.

- **In GLSL**: You'll need to pass the mouse position as a uniform variable:

glsl

Copy code

```
uniform vec4 uMouse; // x, y, z, and w can represent mouse position and state
```

4. **iChannel0**, **iChannel1**, **etc.**: These variables represent textures that you can sample from.

- **In GLSL**: You'll need to pass these textures as uniform sampler2D variables:

glsl

Copy code

```
uniform sampler2D uTexture0; // Use this uniform in your shader
```

5. **mainImage(out vec4 fragColor, in vec2 fragCoord)**: This is the main function in ShaderToy shaders, where **fragColor** is the output color and **fragCoord** is the coordinate of the current fragment.

- **In GLSL**: In a typical GLSL setup, you might use a similar fragment shader function, but the signature and setup might vary depending on your framework. For example, in a typical OpenGL setup, the fragment shader might look like this:

glsl

Copy code

```
void main() {
 // Your code here
}
```

6. **fragCoord.xy**: This variable provides the coordinates of the current fragment.

- **In GLSL**: You can pass these coordinates as a varying variable or compute them based on the screen dimensions:

glsl

Copy code

```
vec2 fragCoord = gl_FragCoord.xy; // In a GLSL shader
```

When porting shaders, make sure to adapt any dependencies on the ShaderToy environment to the specifics of your target environment. This usually involves setting up appropriate uniform variables, handling texture inputs and outputs, and ensuring that your fragment shader logic aligns with the rendering pipeline you're using.