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:
normalized value=original value255\text{normalized value} = \frac{\text{original value}}{255}
normalized value=255original value

For example:

- A pixel with the color `(128, 64, 255)` in integer space would be represented as:
  - **Red**: 128255≈0.502\frac{128}{255} \approx 0.502255128≈0.502
  - **Green**: 64255≈0.251\frac{64}{255} \approx 0.25125564≈0.251
  - **Blue**: 255255=1.0\frac{255}{255} = 1.0255255=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 toanipulate 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.

Revision #7
Created 9 September 2024 22:09:49 by victor
Updated 12 September 2024 22:52:48 by victor