Introduction to OpenGL

Evangelou Iordanis

Presentation co-author Anastasios Gkaravelis

Graphics programming in a nutshell

- A combination of math, computer science and programming
- The goal is to :
  - Describe static or dynamic environments (e.g. 3D scenes)
  - Generate images (e.g. photorealistic rendering)
- The majority of cases exploit a GPU to :
  - Perform efficient and parallel computations
  - Fit in a tight budget of time (60FPS -> ~16ms, 120FPS -> ~8ms)
- Every algorithm can also be employed in a CPU (e.g. scientific work)
- Or design a pipeline that uses the best of both worlds (e.g. game engines)

Why we need a GPU

- Special purpose hardware for :
  - Drawing triangles
  - Parallel processing for general purpose tasks
- Very fast with simple operations or highly parallel processes !
  - Uses hundreds of processing cores to execute instructions
  - Not so fast for other stuff

What we aim for in these courses

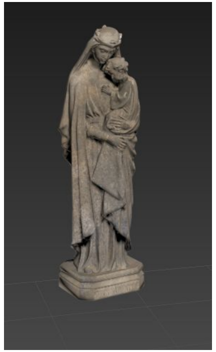
- Gain an insight about the OpenGL API and its capabilities
- Learn how to combine math with graphics programming
- Understand the importance of parallel computing
- Learn how to use the rasterization pipeline for :
  - 3D scene rendering
  - Basic environment lighting effects
  - Basic post processing effects

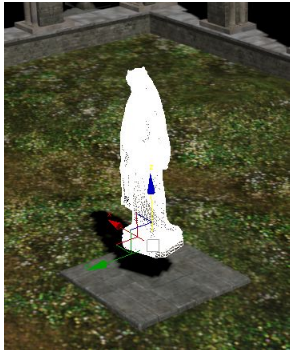
Image synthesis

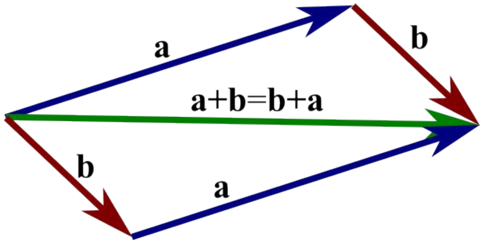
- A high resolution image has millions of pixels (4K - 3840x2160)
- 3 bands for each pixel
  - Specifying red, green and blue color values (0 - 255)

Image synthesis

- In order to render a scene we need:
  - Geometry
  - 3D Coordinates
  - Vector Math







What is a Graphics API

- Common interface for apps to communicate with different GPUs
- Controls the flow of data between CPU - GPU
- Instructs the GPU to execute a specific queue of commands
- Most Common Graphics APIs are:
  - Direct3D
  - OpenGL
  - Vulkan
  - Metal

OpenGL

- High level API mainly for rendering purposes (but not only)
- Operates along with a programmable rendering pipeline through GLSL language
- Implemented in device driver
- Cross-Platform (Windows, Linux, Mac OS, Mobile)
- Hardware-Independent

What OpenGL does not do

- OpenGL is strictly for triggering the GPU cores to execute user specific code
- **Does not** create models
- **Does not** create and handle window context
- **Does not** handle input
- **Does not** handle audio
- **Does not** do vector math

SDL library

- Handles Windows, Input, Video, Audio, Filesystem, Threads
- It provides an interface in C
- Cross-Platform (Windows, Linux, Mac OS, Android, Embedded systems)
- It can be enhanced using:
  - SDL_image: support multiple image formats
  - SDL_mixer: support advanced audio functionality
  - SDL_ttf: support TrueType font rendering

SDL library

- SDL_Init for initializing the SDL
- SDL_Window* SDL_CreateWindow to create a window
- SDL_GL_SetAttribute for setting OpenGL attributes
- SDL_GLContext SDL_GL_CreateContext for creating an openGL context
- SDL_GL_SetSwapInterval to enable/disable V-Sync
- SDL_PollEvent for polling window, keyboard, mouse events
- SDL_GL_SwapWindow to swap buffer when using double buffering
- SDL_GL_DeleteContext/SDL_DestroyWindow/SDL_Quit to destroy states

GLSL programming

- OpenGL Shading Language
- C like syntax
- Write programs that run in parallel in the GPU
- Can program all stages of the Rasterization pipeline
- Strictly input-output
- Useful built-in functions (https://www.khronos.org/registry/OpenGL-Refpages/gl4/)

GLSL programming

- Scalar values:
  - `bool, int, uint, float`
- Vector values:
  - `vec2, vec3, vec4`
  - `ivec2, ivec3, ivec4`
  - `uvec2, uvec3, uvec4`
  - `bvec2, bvec3, bvec3`
  - Vectors can be accessed with operator [] or ({.x .y} for vec2, {.x .y .z} for vec3, etc)
  - Can also perform swizzle operations e.g. :
    - `vec3 b = a.xxy`
    - `vec2 c = a.yx`
- Matrices
  - `mat2, mat3, mat4`
- Structs composed of primitive types :
  - `struct data { int a; vec2 b; };`

GLM library

- Math library dedicated for vector math in CPU
- Replicates the coding style of GLSL for convenience
- Cross platform

GLSL programming

- Download Lab1 from eclass
- The program generates an image by rendering a scene
- We will apply post processing filters in the generated image
- We will alter the color of the final image using a GLSL program

GLSL programming

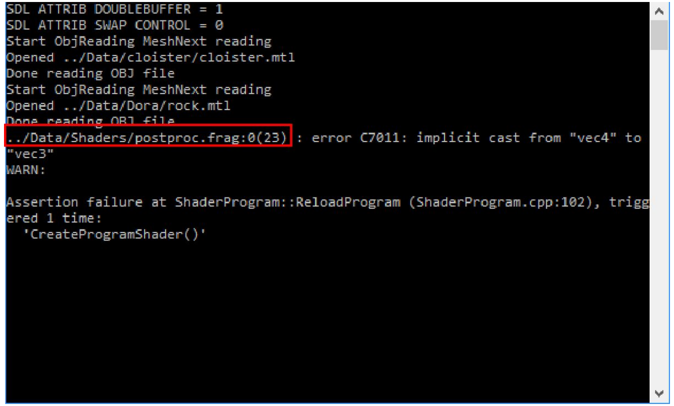
- Compile and run Lab1 (Release mode)
- You can move with *(WASD)* or arrow keys and mouse (left click)
- Open shader Assets\Shaders\postproc.frag

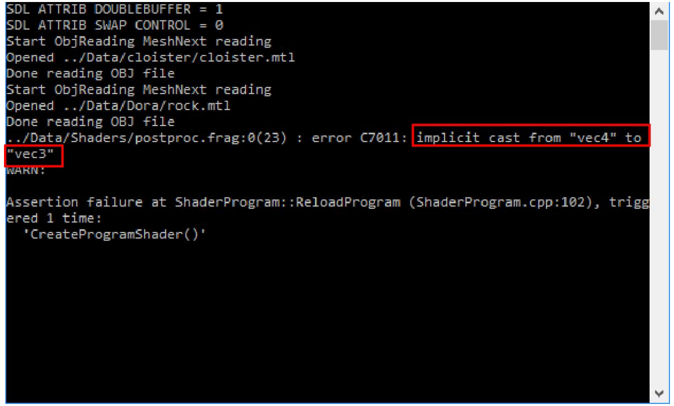
GLSL programming

- The project provides an automatic shader reload function by pressing “R”
- If an error occurred :

GLSL programming

- Prints error messages in console
- The shader where the error occurred and in which line in the shader:





GLSL programming

- Fix the error and press “Retry”

GLSL programming

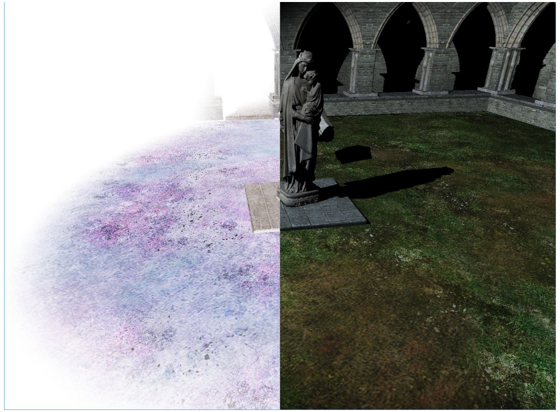
- Each GLSL program runs in parallel, with different input and output
- We execute an independent program for each pixel on the screen
- We spawn as many thread as there are pixels and each thread will write to a different pixel
- We can access the thread’s **pixel** coord using **`gl_FragCoord.xy`**

GLSL programming

- We can access the thread (pixel) coord using `gl_FragCoord.xy`
- Sample the image using `texture(sampler_name, coord)`
  - Where coord is normalized [0, 1] coordinates
- Normalized coordinates are already provided in `vec2 uv;`

GLSL programming

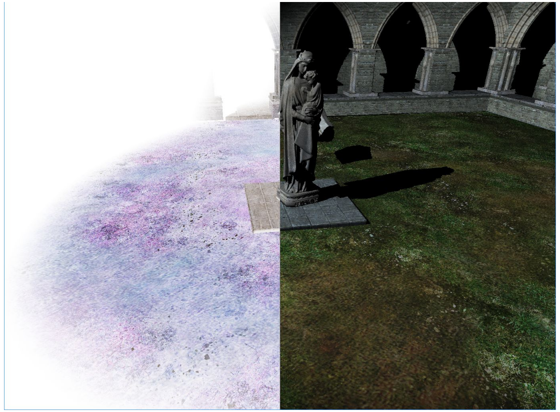
- Sample the texture at the current pixel position
  - `vec3 color = texture(uniform_texture, uv).rgb;`
  - Texture always return a vec4 (RGBA), we only need the RGB components
- Make the left half of the image with negative colors
  - `color = vec3(1.0) - color;`



GLSL programming

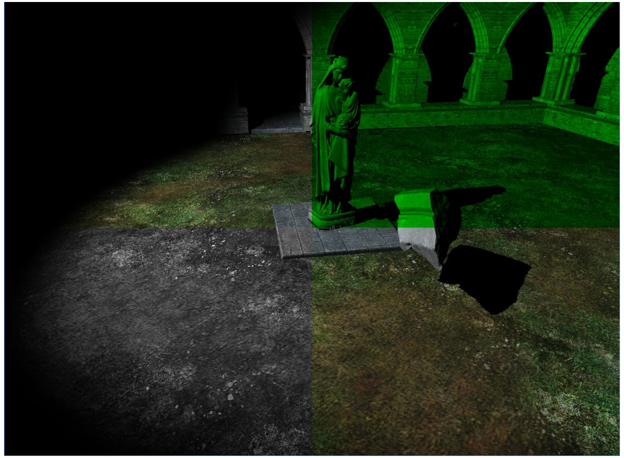
- Sample the texture at the current pixel position
  - `vec3 color = texture(uniform_texture, uv).rgb;`
  - Texture always return a vec4 (RGBA), we only need the RGB components
- Make the left half of the image with negative colors
  - `color = vec3(1.0) - color;`
- Solution:

```
if(uv.x < 0.5)
    color = vec3(1.0) - color;
```



GLSL programming

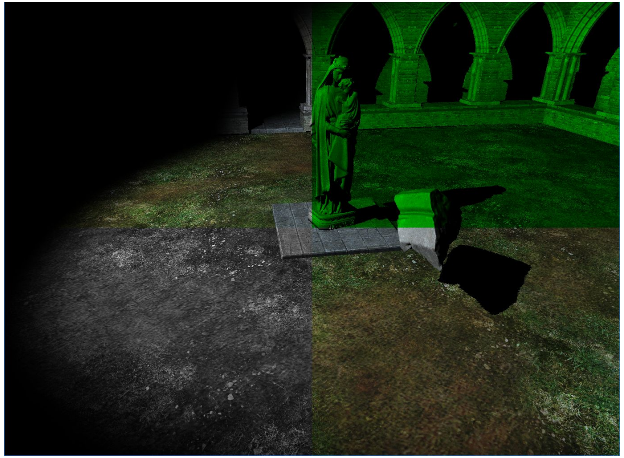
- Make the bottom left quarter grayscale
  - `color = vec3(color.r + color.g + color.b) / 3.0;`
- Keep from the top right quarter only the green component
  - `color = color * vec3(0, 1, 0);`



GLSL programming

- Make the bottom left quarter grayscale
  - `color = vec3(color.r + color.g + color.b) / 3.0;`
- Keep from the top right quarter only the green component
  - `color = color * vec3(0, 1, 0);`
- Solution:

```
if(uv.y < 0.5 && uv.x < 0.5)
    color = vec3(color.r + color.g + color.b) / 3.0;
if(uv.y > 0.5 && uv.x > 0.5)
    color *=  vec3(0,1,0);
```



GLSL programming

- Checkerboard (10x10)

```
if( int(uv.x * 10) % 2  + int(uv.y * 10) % 2 == 1 )
    color = vec3(1,0,0);
```

- Pixelation

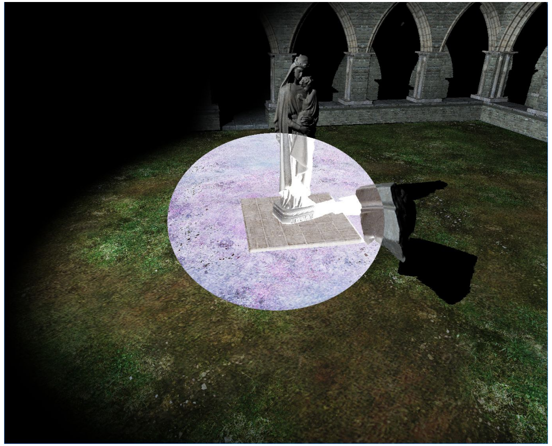
```
ivec2 size = textureSize(uniform_texture, 0).xy;
// sample pixels with a 5 pixel stride
float dx = 5.0*(1./size.x);
float dy = 5.0*(1./size.y);
vec2 coord = vec2(dx*floor(uv.x/dx),
dy*floor(uv.y/dy));
color = texture(uniform_texture, coord).rgb;
```

GLSL programming

- How about rendering a disk shape in the center of the image?
- Recall that:
    - $$D = \left\{ (x,\, y) \in \mathbb{R}^2 : \sqrt{(x - a)^2 + (y - b)^2} \leq r \right\}$$
- How do we adapt the above case to our algorithm ?
    - what indicates parameter **r** ?
    - what indicates parameters **a** and **b** ?
    - what indicates parameters **x** and **y** ?

GLSL programming

- How about rendering a disk shape in the center of the image?
- Recall that:
  - $$D = \left\{ (x, y) \in \mathbb{R}^2 : \sqrt{(x - a)^2 + (y - b)^2} \leq r \right\}$$
- How do we adapt the above case to our algorithm ?
  - what indicates parameter **r** ?
  - what indicates parameters **a** and **b** ?
  - what indicates parameters **x** and **y** ?



- Possible solution* :

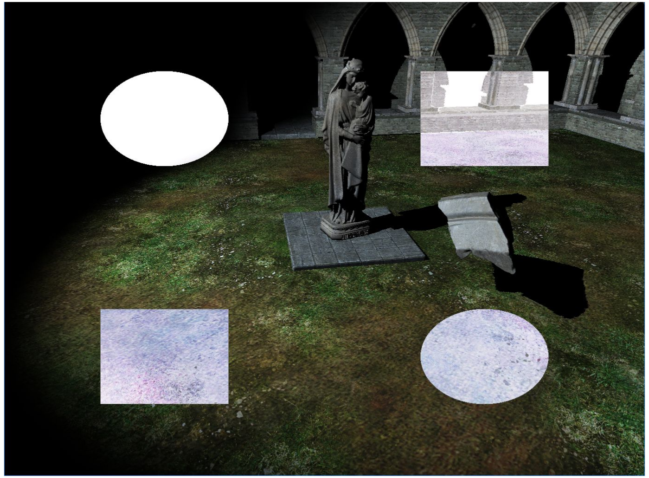
```
vec2 pos = uv - vec2(0.5);
float dist = sqrt(pos.x * pos.x + pos.y * pos.y);

if(dist < 0.2) color = vec3(1.0) - color;
```

GLSL programming

- What about a rectangle?

```
vec2 pos = uv - vec2(0.5);
if(abs(pos.x) < 0.1 && abs(pos.y) < 0.1)
    color = vec3(1.0) - color;
```



- HOME
  - Change center and size of circle and rectangle
  - Put 4 shape in each quarter of the image

GLSL programming

- Let's make a night vision sniper google
- First we need a disk
- For a change let's convert the space from [0, 1] to [-1, 1]
  - `vec2 pos = 2.0 * uv - vec2(1.0);`
  - Now (0,0) is the center of the image
- Create a mask and multiply it with color

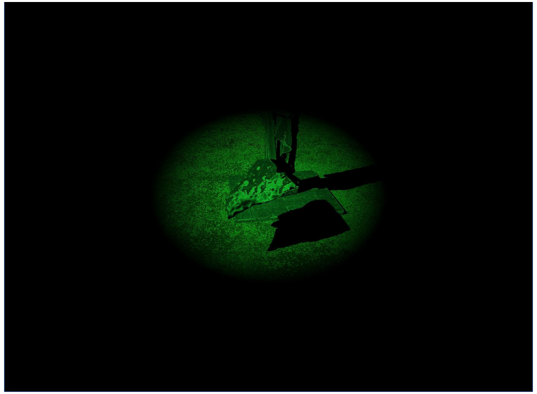
```
float dist = length(pos);
float mask = (dist < 0.3) ? 1 : 0;
color = color * mask;
```

GLSL programming

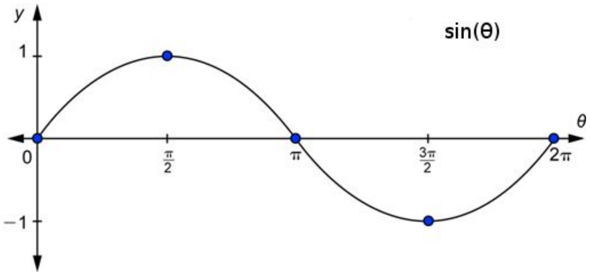
- Use mask with a falloff function
- Radius takes values in [0, 1]
- Make the center visible with a linear falloff function
  - `mask = 1.0 - dist;` // now it takes values in [1, 0]
- Maybe use a quadratic falloff function for a nicer result
  - `mask = 1.0 - (dist * dist);`
- Boost the falloff to make the circle smaller
  - `mask = 1.0 - (dist * dist) / 0.2;`

GLSL programming

- Blend using a "green" filter
  - `vec3 visionColor = vec3(0.1, 0.95, 0.2);`
  - `color = color * visionColor * mask;`
- Maybe boost pixels with little luminance
  - Compute the luminance value of the pixel
    - `float lum = dot(vec3(0.30, 0.59, 0.11), color);`
  - Boost pixel luminance
    - e.g. `color *= 4.0;`



GLSL programming



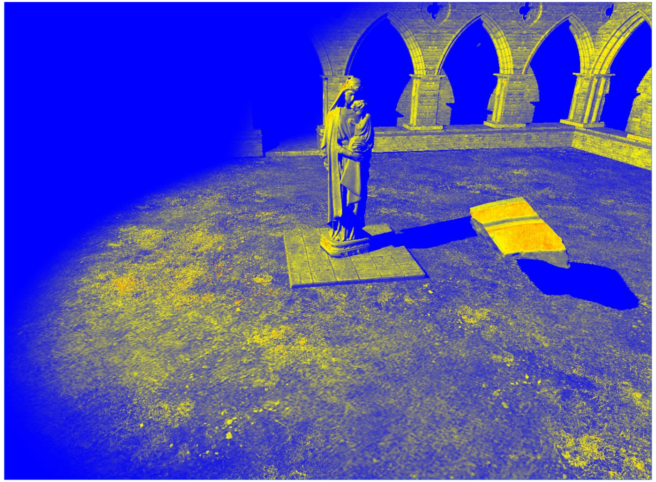
- What about making it move??
- We can use a periodic function
    - We can use the sine function to move up-down the circle
    - `pos.y += 0.4*sin(uniform_time);`
- What about move in a circular motion?
    - Recall the identity $\cos^2\theta + \sin^2\theta = 1$
    - `pos += 0.4 * vec2(sin(uniform_time), cos(uniform_time));`
- Maybe move in an ellipse??
    - Just squeeze one dimension of the circle
    - `pos += 0.4 * vec2(0.5 * sin(uniform_time), cos(uniform_time));`

GLSL programming

Bonus: “Predator” heat vision

- Compute the luminance value of the pixel
- If the luminance is below 0.5, linear blend between blue and yellow
- If the luminance is above 0.5, linear blend between yellow and red

```
vec3 colors[3];
colors[0] = vec3(0.,0.,1.);
colors[1] = vec3(1.,1.,0.);
colors[2] = vec3(1.,0.,0.);
float lum = dot(vec3(0.30, 0.59, 0.11), color.rgb);
float ix = (lum < 0.5)? 0.0 : 1.0;
vec3 a = mix(colors[0], colors[1], ix);
vec3 b = mix(colors[1], colors[2], ix);
color = mix(a, b, (lum - ix * 0.5) / 0.5);
```



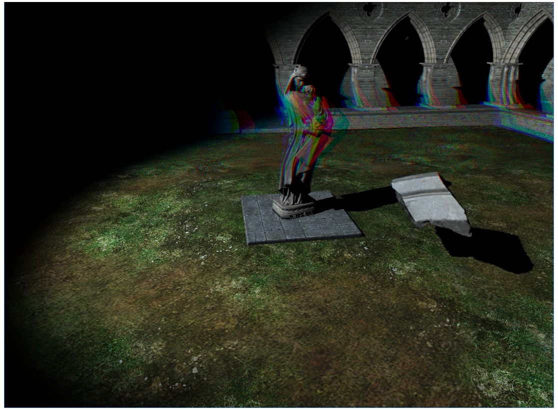
GLSL programming

Bonus: “The Matrix” mirror effect

- To distort the image sample from a different coordinates
- Sample in the direction away from the center of the image
- Choose the sampling position based on a cosine function

```
vec2 p = 2.0 * uv - 1.0;
float len = length(p);
float frequency = 12.0;
float speed = 2.0;
// direction away from the center
vec2 direction = p / len;
// add to UV, the direction scaled by cosine() * 0.02
uv = uv + direction * cos(len * frequency - uniform_time * speed) * 0.02;
color = texture2D(uniform_texture, uv).xyz;
```

GLSL programming



Bonus: Chromatic Aberration (old TV)

- Sample each color component from a different pixel
- Change the position using in the Y axis using elapsed time
- Change the amplitude of the effect based on an ***exponential*** function

```
float dist = 1.0 - mod(0.3 * uniform_time, 1.0) - uv.y;
dist = dist * dist;
float time = exp(1 - 200 * dist) / 2.8;
float power = time;
power *= 0.6;
float red = texture(uniform_texture, uv + power * vec2(-0.1, 0)).r;
float green = texture(uniform_texture, uv + power * vec2(-0.05, 0)).g;
float blue = texture(uniform_texture, uv + power * vec2(-0.025, 0)).b;
color = vec3(red, green, blue);
```

Next labs

We will learn how to render the previous scene step by step!!!