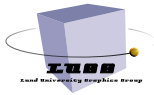
Introductory Seminar

EDAF80: Computer Graphics

Rikard Olajos



LUND UNIVERSITY
LTH
FACULTY OF ENGINEERING

AGENDA

1. Lab info
2. OpenGL
3. C++ crash course

LABS OVERVIEW

- 1 optional + 5 mandatory assignments
  - Week 2 – 7
  - "Lab 0" in week 2: optional attendance
  - Book sessions on course homepage
- Work in pairs
  - If looking for a partner, post on forum
- E:Uranus
  - Located in E-huset basement
  - Windows 10, 64-bit, Core i5, 8GB RAM
  - Visual Studio 2022
  - Geforce GTX 560

OPENGL

- Application Programming Interface (API)
  - Set of functions that create a 2D image of a 3D scene
  - 3D scene is made of:
    - Primitives – Triangles
    - Textures – 2D images
    - and much more!
- Controls a graphics pipeline (graphics hardware)
  - Graphics Processing Unit (GPU)

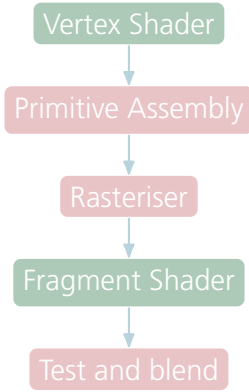
OPENGL

- We will focus on the core profile
  - no fixed function/immediate mode
- OpenGL is a state machine
  - Current state is the “OpenGL context”
  - There are many functions that change the current state
  - OpenGL uses objects that are a part of the state
  - Drawing uses the most recently bound buffers

APPLICATION SETUP

- First, make a window, use GLFW library
- Second, create a `while` loop i.e. the render loop:
  - Grab inputs
  - Render the screen
  - Swap the buffers
- Third, do some rendering in the render loop...

GRAPHICS PIPELINE



- Shaders are programmable, other parts are not
- There are no default vertex and fragment shaders, you must provide them
- Primitive Assembly (PA) puts the vertices into the primitive that is currently specified

VERTICES

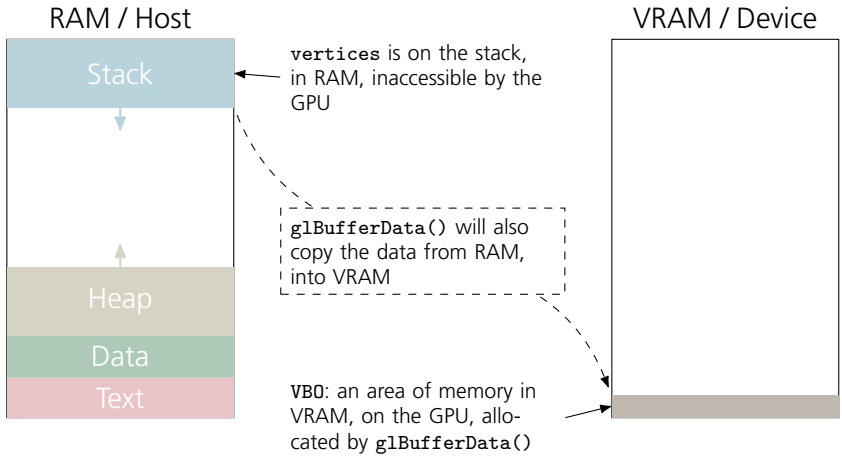
- 3 vertices in (x, y, z)
  - Range is [-1, +1]

```
GLfloat vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```

- Output from VS is in Normalized Device Coordinates (NDC)
  - Also [-1, +1]
  - Origin is in the middle of the screen
- Put vertices into Vertex Buffer Objects (VBO)

```
GLuint VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

WHERE ARE YOUR VERTICES?



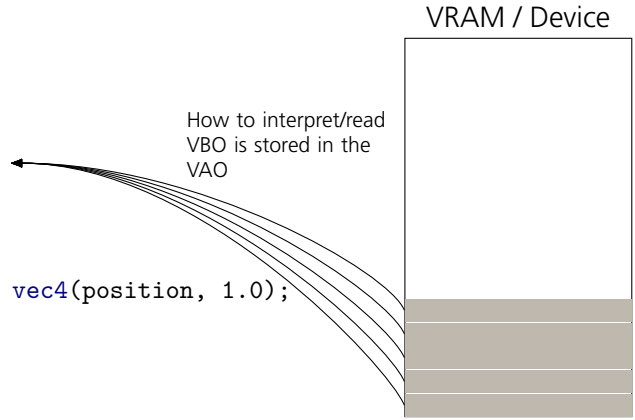
SIMPLE VERTEX SHADER

- Must set the predefined variable `gl_Position`
- Need to link vertex data to the vertex shader
  - A Vertex Array Object (VAO) is also required

```
#version 410

in vec3 position;

void main()
{
    gl_Position = vec4(position, 1.0);
}
```

HOW TO ACCESS THE VERTICES

```
#version 410

in vec3 position;

void main()
{
    gl_Position = vec4(position, 1.0);
}
```



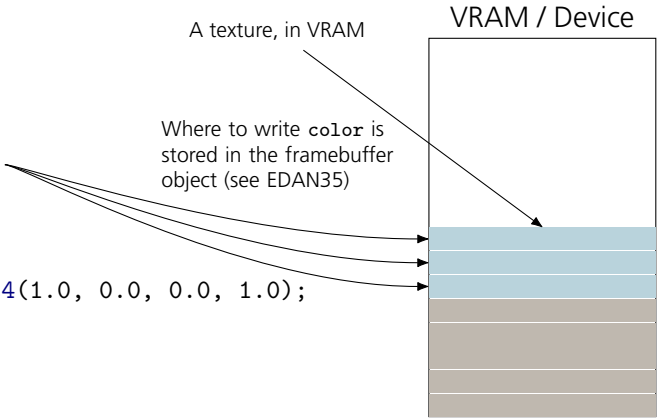
SIMPLE FRAGMENT SHADER

- Requires one output variable of `vec4`, for the colour

```
#version 410

out vec4 color;

void main()
{
    color = vec4(1.0, 0.0, 0.0, 1.0); // set color to red
}
```

WHERE DOES FRAGMENT OUTPUT GO?

```
#version 410

out vec4 color;

void main()
{
    color = vec4(1.0, 0.0, 0.0, 1.0);
}
```



BUFFERS VERSUS TEXTURES

- Both reside in VRAM on the GPU
- Both represent a chunk of memory (both can be viewed as ***n***-d arrays, with data in cells)

Buffers:

- Supports any data format (even custom)
- Only the cells can be read
- Stored linearly

Textures:

- Only specific data formats allowed
- You can read between cells, and get interpolated results
- Stored in tiles

COMPILING SHADERS

- Shaders run on the **GPU**, not CPU
- They are written in GLSL, which is C-based
- Like for CPUs, need to compile to machine-specific instructions
- Unlike CPUs, shader compilation is done at runtime by your GPU driver

COMPILING SHADERS

Done in two steps:

1. Compile each shader individually

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
...
```

   - Check for possible compile errors after `glCompileShader()`

2. Link all shaders into a single shader program

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);
...
```

   - Check for possible linking errors after `glLinkProgram()`

DRAWING

- Tell OpenGL what to render
  - `glDrawArrays(GL_TRIANGLES, 0, 3);`
  - (what to draw, starting index, number of vertices)

MORE INFO

- `https://learnopengl.com/`
  - Joey de Vries
  - Most complete guide for modern OpenGL
- `https://open.gl/`
  - Alexander Overvoorde
- `https://antongerdelan.net/opengl/`
  - Anton Gerdelan

HELLO C++

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello world!\n";
    return 0;
}
```

Output

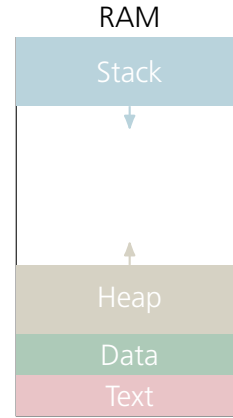
```
> Hello world!
```

ABOUT

- Based on C
- Create by Bjarne Stroustrup in 80's 🇩🇰
- Object-oriented (classes and structs)
- Constructors & destructors
- Inheritance & virtual functions
- Operator overloading (+, -, *, /, etc.)
- Templates
- C++11 began a 3-year cycle of updates

SIMPLIFIED MEMORY MODEL

- Stack
  - Stores local variables
  - Managed by the compiler
- Heap
  - Dynamic memory
  - Managed by the programmer
- Data
  - Stores global variables
  - Initialized and uninitialized
- Text
  - Stores code being executed



STACK INTEGER DECLARATION

```
int x;

std::cout << x;
```

Output

```
> 698683442
```

STACK INTEGER DECLARATION & INITIALIZATION

```
int x;
x = 35;

std::cout << x;
```

Output

```
> 35
```

POINTER TO AN INTEGER

```
int* y;

std::cout << y;
```

Output

```
> 0000000B2394FB09
```

ALLOCATE HEAP MEMORY

```
int* y;
y = new int(10);

std::cout << y;
```

Output

```
> 000001D7AD807FA0
```

POINTER DEREFERENCING

```
int* y;
y = new int(10);

std::cout << *y;
```

Output

```
> 10
```

`*y` is dereferencing the pointer

POINTER TO STACK INTEGER

```
int x = 35;
int* xp;
```

POINTER TO STACK INTEGER

```
int x = 35;
int* xp = x; // Wrong!
```

`x` is an `int`, not an `int*` (pointer to an int)

POINTER TO STACK INTEGER

```
int x = 35;
int* xp = &x;
```

`&x` takes the address of `x`

```
std::cout << *xp;
```

Output

```
> 35
```

HEAP DEALLOCATION

```
int* y = new int(10);
…
delete y;
```

C++ CLASS

```
class MyClass
{

};
```

CLASS SCOPE

```
class MyClass
{
    // class scope
};
```

CLASS ACCESS SPECIFIERS

```
class MyClass
{
private:
    // access within this class only (default)
protected:
    // access to this and inherited classes
public:
    // access to everyone
};
```

CLASS CONSTRUCTOR

```
class MyClass
{
    float mX;

    MyClass(float x)
    {
        mx = x;
    }
};
```

CLASS CONSTRUCTOR + INITIALIZATION

```
class MyClass
{
    float mX;

    MyClass(float x) : mX(x)
    {

    }
};
```

CLASS CONSTRUCTOR & DESTRUCTOR

```
class MyClass
{
    float mX;

    MyClass(float x)
    {
        mX = x;
    }

    ~MyClass()
    {
        // mX is on stack, so automatically deallocated
    }
};
```

CLASS CONSTRUCTOR & DESTRUCTOR

```
class MyClass
{
    float* mXp;

    MyClass(float x)
    {
        mXp = new float(x);
    }

    ~MyClass()
    {
        delete mXp; // mX is on heap, so deallocate manually
    }
};
```

CLASS MEMBER METHOD

```
class MyClass
{
    float mX;

    void setX(float x)
    {
        mX = x;
    }
};
```

CLASS MEMBER ACCESS

Stack

```
MyClass myclass = MyClass(5);
myclass.setX(2);
```

Heap

```
MyClass* myclassp = new MyClass(5);
myclass->setX(2);
…
delete myclassp;
```

CLASS DECLARATION + DEFINITION

MyClass.h

```
class MyClass {
    float mX;
    void setX(float x);
};
```

MyClass.cpp

```
#include "MyClass.h"

void MyClass::setX(float x) {
    mX = x;
}
```

ARRAY: STACK & HEAP ALLOCATION

Stack

```
float numbers[3];
numbers[0] = 1.0f;
…
```

Stack: direct initialization

```
float numbers[3] = { 1.0f, 2.0f, 3.0f };
```

Heap

```
float* numbers = new float[3];
number[0] = 1.0f;
…
delete[] numbers;
```

PARAMETERS: VALUE, REFERENCE, POINTER

```
MyClass mc0 = MyClass(1);
MyClass mc1 = MyClass(1);
MyClass* mc2 = new MyClass(1);

foo(mc0, mc1, mc2);
```

```
int foo(MyClass mc0, MyClass& mc1, MyClass* mc2)
{
    mc0.setX(10);  // edits local copy only
    mc1.setX(10);  // edits original
    mc2->setX(10); // edits original
}
```

TYPES

```
int                a = -1;              (32 bits)
unsigned int       b = 1u;              (32 bits)
long               c = -2l;             (64 bits)
unsigned long      d = 2lu;             (64 bits)
float              e = 1.0f;            (32 bits)
double             f = 3.14;            (64 bits)
bool               g = true;            (8 bits)
char               h = 'x';             (8 bits)
char               i[] = "abcd";        (5 * 8 bits)
…                  …                    …
```

OPERATOR OVERLOADING

- May customize +, -, *, /, and many others
- Very useful for linear algebra, e.g.:

```
glm::mat3 A, B;
glm::vec3 u;
…
glm::mat3 M = A * B;
glm::vec3 v = M * u;
```

OUTPUT

- Print "Rendering..." to standard output, followed by a new line:

```
std::cout << "Rendering...\n";
```

- Or, with the same result:

```
printf("Rendering...\n");
```

- Inclusion of variables (many formatting options available):

```
std::cout << "an integer: " << 1 << ", a float: " << 3.14f << '\n';
```

- Or, with the same result:

```
printf("an integer: %d, a float: %f\n", 1, 3.14f);
```

MORE INFO

- EDAF50 – C++ Programming
- `https://cplusplus.com/`
- `https://en.cppreference.com/w/cpp`