COMP3421/9415
Computer Graphics

Introduction

Robert Clifton-Everest

Email: robertce@cse.unsw.edu.au
Administriva

- **Who:** Robert Clifton-Everest (lecturer), Ali Darejeh (admin)

- **Where:** [http://www.cse.unsw.edu.au/~cs3421](http://www.cse.unsw.edu.au/~cs3421)
  - Same website for COMP9415

- **What:** See the course outline
Lectures

• Lecture videos are linked from the course website

• Timetable is a bit complicated

• Lecture starter code is released before each lecture

  - Code along if you want
Lab

- Optional lab this week (not marked)
- Attend any session you like
- Opportunity to get your laptop setup for the practical components of the course
- Thursday 3-4PM or Friday 2-3PM in piano lab (K14, behind physics theatre)
Tutorials

• Tutorials start this week!

  - Reenforce what we cover in the Lectures
  - You’ll need to pick an assignment partner for the second assignment, so it’s a good idea to get to know people!
Assignments

• Assignment 1
  - Individual
  - 2D graphics
  - Due at the end of week 4

• Assignment 2
  - Pairs
  - 3D graphics
  - Milestone 1 due at end of week 7
  - Milestone 2 due at the end of week 10
  - Demonstrate in week 11
Quizzes

- 5 online quizzes throughout the course
- Released in weeks 1, 3, 5, 7 and 9
- Due at the end of weeks 2, 4, 6, 8, and 10
Assumed knowledge

• Java
  - Don’t be afraid to ask questions

• Basic linear algebra
  - Vectors, matrices
  - We will revise this
Gained knowledge

• Computer graphics (obviously)

• We also touch on many other areas
  - Linear algebra
  - Geometry
  - High-performance computing
  - Parallelism
  - Software engineering
Why Graphics?

• Games

• Movies and TV

• Visualisations

• Something else?
What will you create?
How?

• Algorithms to automatically render images from models.
How?

• Based on:
  - Geometry
  - Physics
  - Physiology/Neurology/Psychology

• A lot of simplifications and hacks to make it tractable and look good.
CPU vs GPU
CPU vs GPU

- CPU consists of a few cores optimized for sequential serial processing
- GPU has a massively parallel architecture (SIMT/Single Instruction Multiple Thread) consisting of smaller special purpose cores designed for parallel work.
nums[i] = nums[i]*nums[i];

if (nums[i] % 2 == 0) {
    nums[i] = nums[i] + 1;
} else {
    nums[i] = 0;
}
...

nums = \begin{array}{c}
\text{0} & \text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} & \text{9} \\
\end{array} \\
i \text{ is different for each thread}
nums[i] = nums[i] * nums[i];

if (nums[i] % 2 == 0) {
    nums[i] = nums[i] + 1;
} else {
    nums[i] = 0;
}

...
nums[i] = nums[i] * nums[i];

if (nums[i] % 2 == 0) {
    nums[i] = nums[i] + 1;
} else {
    nums[i] = 0;
}

...
nums[i] = nums[i] * nums[i];

if (nums[i] % 2 == 0) {
    nums[i] = nums[i] + 1;
} else {
    nums[i] = 0;
}

...
nums[i] = nums[i]*nums[i];

if (nums[i] % 2 == 0) {
    nums[i] = nums[i] + 1;
} else {
    nums[i] = 0;
}

...
nums[i] = nums[i] * nums[i];

if (nums[i] % 2 == 0) {
    nums[i] = nums[i] + 1;
} else {
    nums[i] = 0;
}

...
nums[i] = nums[i]*nums[i];

if (nums[i] % 2 == 0) {
    nums[i] = nums[i] + 1;
} else {
    nums[i] = 0;
}

...
nums[i] = nums[i]*nums[i];

if (nums[i] % 2 == 0) {
    nums[i] = nums[i] + 1;
} else {
    nums[i] = 0;
}

...
OpenGL

- A low-level 2D/3D graphics API.
  - Free, Open source
  - Cross platform (incl. web and mobile)
  - Highly optimised
  - Designed to use GPUs
  - We will be using OpenGL
DirectX

• Direct3D
  - Microsoft proprietary
  - Only on MS platforms or through emulation (Wine, VMWare)
  - Roughly equivalent features
Vulcan

- Next generation graphics API
  - Still fairly new
  - Even more low-level than OpenGL
  - Only limited support on some platforms (e.g. Mac)
  - Not quite ready for teaching yet, but hopefully soon
Do it yourself

• Generally a bad idea:
  - Reinventing the wheel
  - Numerical accuracy is hard
  - Efficiency is also hard
  - Hardware variations
Low-level graphics

- OpenGL is used to:
  - transfer data to the graphics memory
  - draw primitive shapes (points, lines, triangles, …) using that data

- More complex things like curves, composite shapes, etc. we have to implement ourselves
  - Composing primitives
  - Running programs (shaders) on the GPU
High-level graphics

• Game engines - Unity, Unreal engine
• Modelling - Maya, Blender, 3DS Max
• CAD
• Microsoft Paint?
The plan

• Learn about techniques, concepts and algorithms relating to computer graphics.

• Use them to implement a high-level graphics library
  - In lectures, tutes, assignments
  - Using OpenGL for the low-level components
• A small high-level graphics library

  • Only VERY basic features (week 1)
  • We will explore and extend it throughout the course
  • Contains some example programs
• A Java library

• A wrapper around OpenGL (a C library)

• Contains NEWT, a basic windowing toolkit

• http://jogamp.org/jogl/www/
• Implementation of the API provided by the GPU driver

• We don’t know how it works internally
• For this course we will focus on how to use it, not the hardware architecture
The lab contains instructions for setting up UNSWgraph and running an example program.

Short version: It is packaged as an eclipse project, so can be directly imported into eclipse with minimal hassle.

NOTE: Doesn’t work on VLAB
My first graphics program

• See HelloDot.java

• Shows ALL features of UNSWgraph version 0.1
Application

• Applications have a single NEWT window

• 2D applications give a simple 2D canvas to draw on.

• The size of the window is given to the constructor.
public class HelloDot extends Application2D {

    public HelloDot() {
        super("HelloDot", 600, 600);
    }

    public static void main(String[] args) {
        HelloDot example = new HelloDot();
        example.start();
    }

    @Override
    public void display(GL3 gl) {
        super.display(gl);
        Point2D point = new Point2D(0f, 0f);
        point.draw(gl);
    }
}
public class HelloDot extends Application2D {

    public HelloDot() {
        super("HelloDot", 600, 600);
    }

    public static void main(String[] args) {
        HelloDot example = new HelloDot();
        example.start();
    }

    @Override
    public void display(GL3 gl) {
        super.display(gl);
        Point2D point = new Point2D(0f, 0f);
        point.draw(gl);
    }
}

Viewport

• We talk in general about the **viewport** as the piece of the screen we are drawing on.

• It may be a window, part of a window, or the whole screen. (In UNSWgraph by default it is the whole window – minus the border)

• It can be any size but we assume it is always a rectangle.

• It has its own coordinate system
Coordinate system

- By default the viewport is centred at (0,0). The left boundary is at x=-1, the right at x=1, the bottom at y=-1 and the top at y=1.
public class HelloDot extends Application2D {

    public HelloDot() {
        super("HelloDot", 600, 600);
    }

    public static void main(String[] args) {
        HelloDot example = new HelloDot();
        example.start();
    }

    @Override
    public void display(GL3 gl) {
        super.display(gl);
        Point2D point = new Point2D(0f, 0f);
        point.draw(gl);
    }
}

Event-based Programming

• UNSWgraph and NEWT are event-driven.

• This requires a different approach to procedural programming:
  - The main() method create an instance of the application and calls start(), which doesn’t terminate.
  - Events are dispatched by the event loop.
  - Handlers are called when events occur.
    • e.g. display() is called 60 times a second
But what’s really going on?

• See Point2D.draw()

• In the draw method for point we have to do 4 main things
  - Create a buffer in main memory containing the point coordinates
  - Transfer that buffer to GPU memory
  - Tell the GPU to draw that buffer as a point
  - Free the buffer in GPU memory
GL3

- GL3 provides access to all the normal OpenGL methods and constants.
  - [http://jogamp.org/deployment/v2.2.4/javadoc/jogl/javadoc/javax/media/opengl/GL3.html](http://jogamp.org/deployment/v2.2.4/javadoc/jogl/javadoc/javax/media/opengl/GL3.html)
- A GL3 object can’t be constructed, cloned or copied in any way
- We have to pass it through to the methods that need it
We have two memory spaces

Main Memory

GPU Memory
Point2DBuffer buffer = new Point2DBuffer(1);

Create a buffer that can store 1 point
The buffer is pinned in main memory.
buffer.put(0, this);

Store the value of this point at index 0 in the buffer

buffer

(x,y)

Main Memory

GPU Memory
int[] names = new int[1];
gl.glGenBuffers(1, names, 0);

Create a new name for a buffer
gl.glBindBuffer(GL.GL_ARRAY_BUFFER, names[0]);

This is the buffer we want to use. All future buffer operations will be on this buffer.
void glBindBuffer(int target,  // Binding target
                 int buffer);  // Name of buffer
Buffer targets

- OpenGL can only have one active buffer of a particular target
- Binding a buffer to GL_ARRAY_BUFFER tells OpenGL that all future operations on the GL_ARRAY_BUFFER are for this buffer
- The GL_ARRAY_BUFFER target is a general purpose target
- Other buffer targets we will see in later weeks.
gl.glBufferData(GL.GL_ARRAY_BUFFER, 2 * Float.BYTES, buffer.getBuffer(), GL.GL_STATIC_DRAW);

This allocates the buffer in graphics memory and transfers the data from main memory into it.
void glBufferData(
    int target,       // Destination
    long size,        // Transfer size (in bytes)
    Buffer data,      // Source
    int usage);       // How it is used
Buffer usage hints

• When allocating a buffer OpenGL lets you give a hint how it might be used.

• OpenGL is free to ignore this information but may use it to optimise how and where it stores the data.

• The most common hints are:

  - GL_STATIC_DRAW — Data will be modified once and used many times

  - GL_DYNAMIC_DRAW — Data will be modified repeatedly and used repeatedly
gl.glBufferData(GL.GL_ARRAY_BUFFER, 2 * Float.BYTES, buffer.getBuffer(), GL.GL_STATIC_DRAW);

Transfer data into the current GL_ARRAY_BUFFER

Main Memory

GPU Memory
gl.glBufferData(GL.GL_ARRAY_BUFFER, 2 * Float.BYTES, buffer.getBuffer(), GL.GL_STATIC_DRAW);

We are transferring $2 \times 4 = 8$ bytes of data.
Using this buffer as a source

```c
gl.glBufferData(GL.GL_ARRAY_BUFFER, 2 * Float.BYTES, buffer.getBuffer(), GL.GL_STATIC_DRAW);
```

http://docs.gl/gl3/glBufferData

Main Memory

GPU Memory
We aren’t going to update the buffer again and it will be used for drawing to the screen.
gl.glVertexAttribPointer(Shader.POSITION, 2, GL.GL_FLOAT, false, 0, 0);

Tell OpenGL that the buffer contains \texttt{vertex} positions.
In OpenGL, a vertex (plural: vertices) is a point that forms part of the definition of a geometric shape. For example:

- 1 vertex defines a point
- 2 vertices define a line
- 3 vertices define a triangle
- 4 vertices *can* define a quadrilateral

Vertices can have attributes attached to them.
void glVertexAttribPointer(
    int index, // The attribute
    int size, // attribute size
    int type, // Primitive type
    boolean normalized, // Normalize ints
    int stride, // Padding
    long pointer_buffer_offset); // Start
The buffer contains the position of the vertices.

```c
// Main Memory
buffer

// GPU Memory
GL_ARRAY_BUFFER

Shader.POSITION

http://docs.gl/gl3/glVertexAttribAttribPointer

gl.glVertexAttribAttribPointer(Shader.POSITION, 2, GL.GL_FLOAT, false, 0, 0);
```
Each position has 2 floats associated with it.

```c
gl.glVertexAttribPointer(Shader.POSITION,
    2, GL.GL_FLOAT, false, 0, 0);
```

http://docs.gl/gl3/glVertexAttribPointer
$$\text{gl.glDrawArrays(GL.GL_POINTS, 0, 1);}$$

Draw the buffer as a point on the screen

![Diagram](http://docs.gl/gl3/glDrawArrays)

Main Memory

GPU Memory
void glDrawArrays(int mode,  // Primitive to draw
                 int first,  // Starting vertex
                 int count);  // Number of vertices
gl.glDeleteBuffers(1, names, 0);

Delete the buffer in graphics memory

http://docs.gl/gl3/glDeleteBuffers
void glDeleteBuffers(int n,
    int[] buffers,
    int buffers_offset);
OpenGL recap

• It is not Object-Oriented, despite us accessing it from Java
  - Use of ints instead of enums
  - Lots of effectively global state

• UNSWgraph is setup to try and report OpenGL errors, but in many cases failure is still silent (e.g. out of bounds errors)

• Error messages can be hard to decipher

• Need to rely on documentation
Questions

• What does it mean when we say OpenGL is low-level?
• Can you remember all the arguments to glVertexAttribPointer?
• Isn’t programming like this really tedious?
From points to lines

- See Line2D.java and HelloLine.java