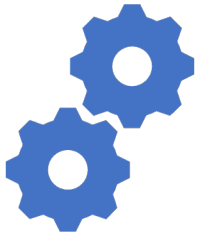
Parallel Implementation of the OpenGL Pipeline (A1)

Dr. Hamish Carr & Dr. Rafael Kuffner dos Anjos

Agenda

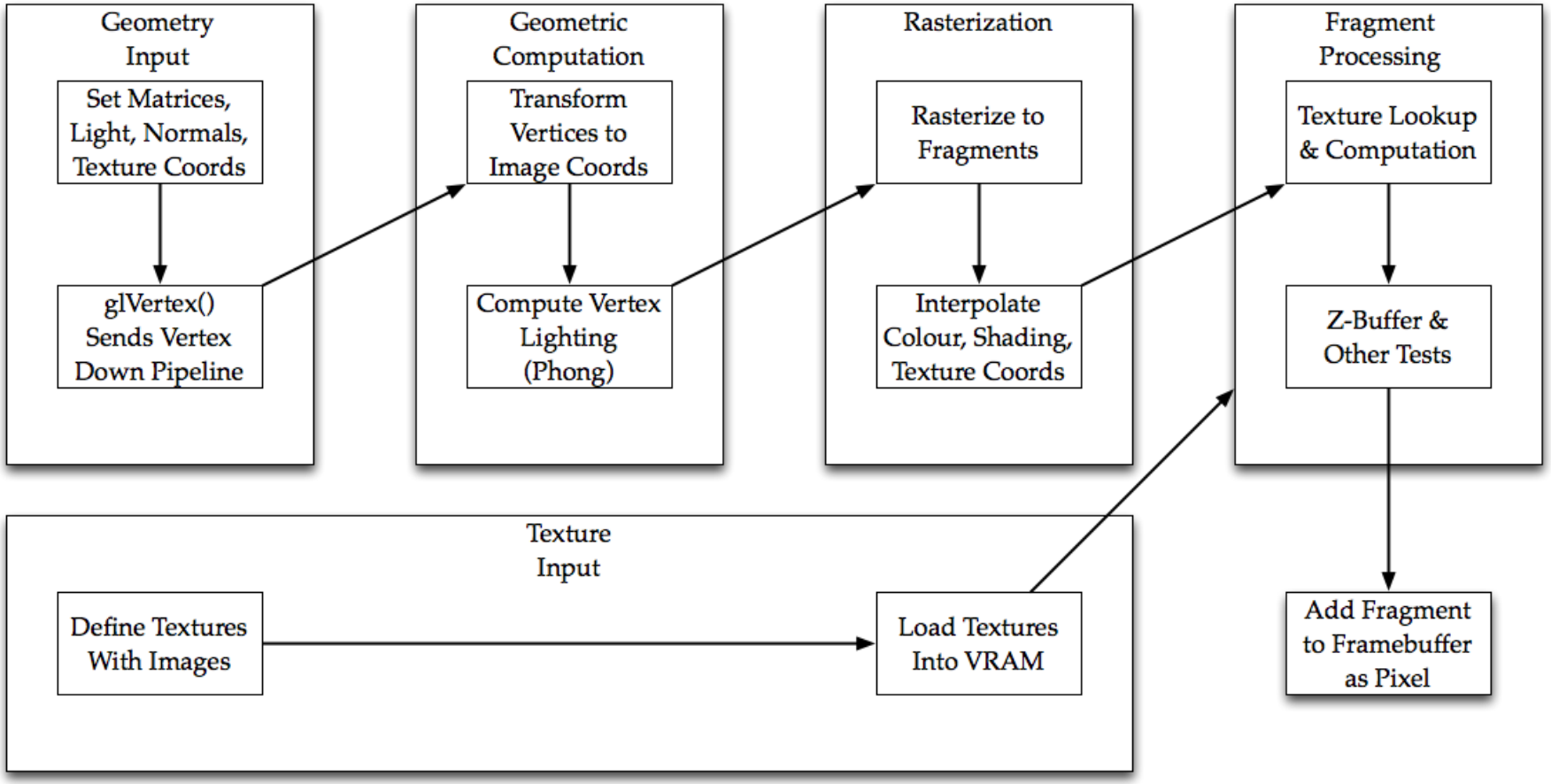
The projective Pipeline



Parallel implementation

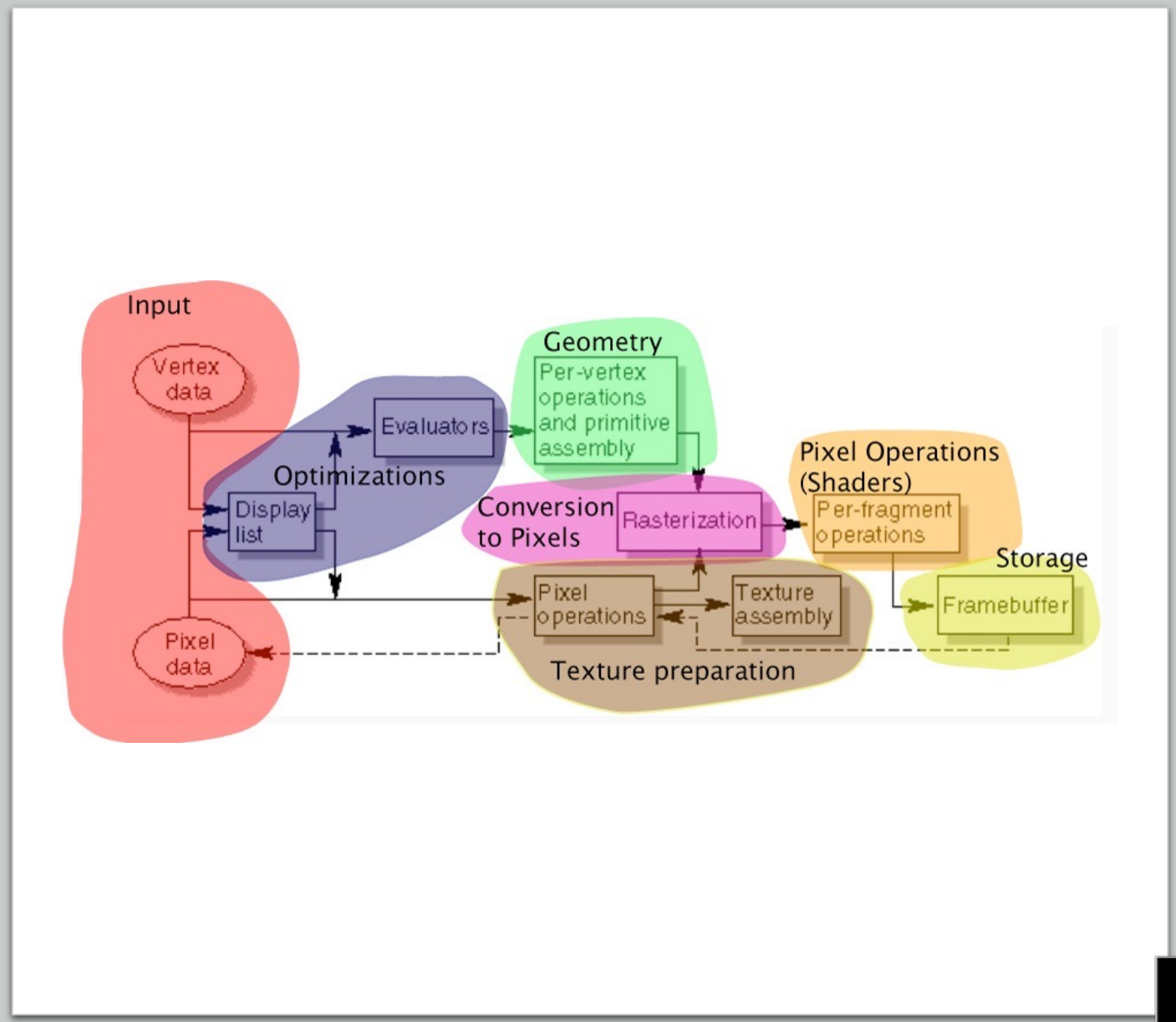
Assignment 1

The Projective Pipeline



Fixed Function Pipeline

- This is the *original* hardware design
- Deeply pipelined, massively parallel
- Now simulated by GP processors

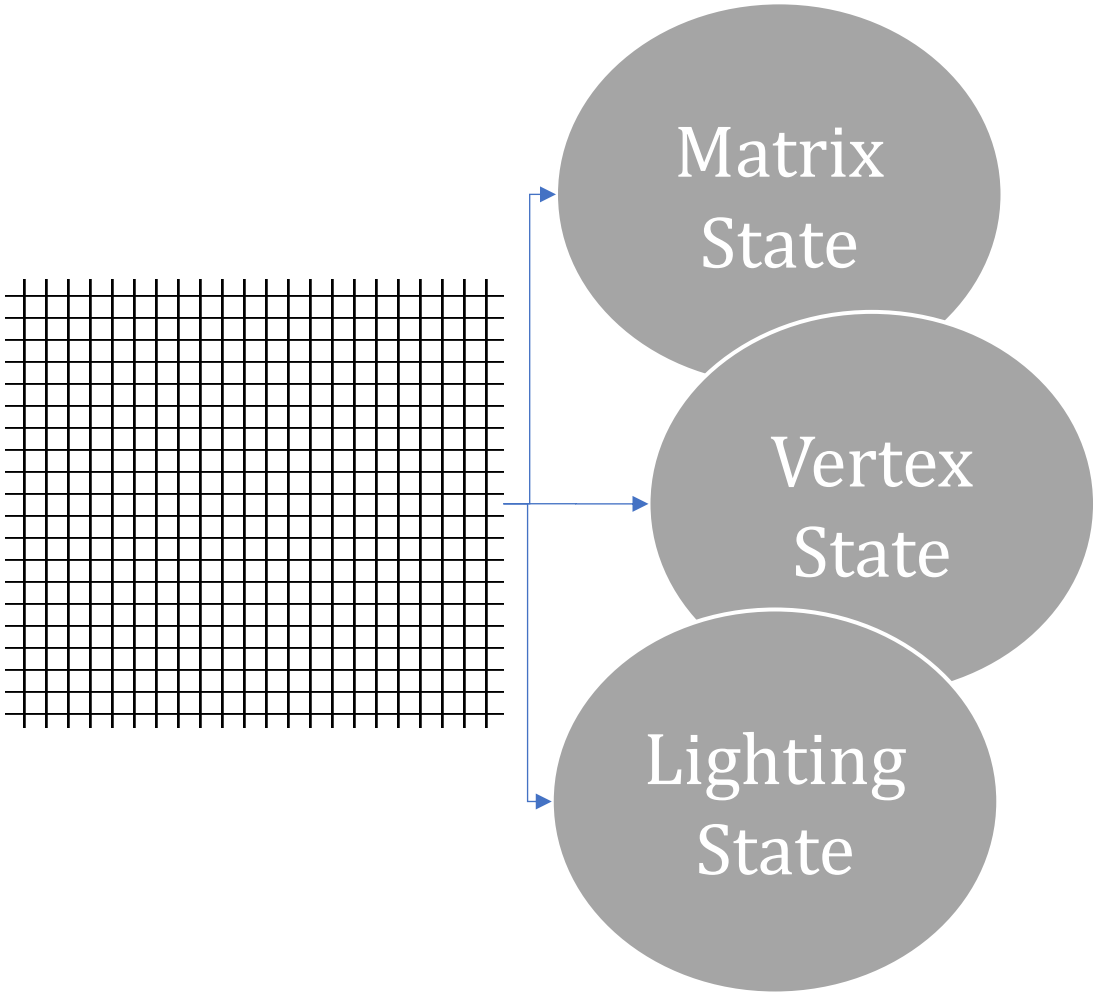


Pipeline Stages

- Input: Vertex(), Normal(), &c.
- Transform: TransformVertex()
- Raster: RasterisePrimitive()
- Fragment: ProcessFragment()
- Framebuffer: Clear()
- Texture Prep: Texture()
- State Changes: all the other functions

States

- Pipeline state is set by function calls
- Define variables for each piece of state



Queues

- Data processing is via queues
- Stages read from one queue, write to another
- Input stage does not read from a queue
- Fragment stage writes directly to image

When/how do we invoke stages?

What happens when we change state?

What does flushing the pipeline do?

How do we prevent collisions on queues?

How do we know when we are done?

Parallel Issues

When/how do we invoke stages?

A1 hint!

1- **FakeGL::Vertex() trigger** later stages

2- Have an EndFrame() routine trigger stages

3- Use a separate thread for each stage

4- Use multiple threads for each stage

Use multiple threads & queues for each stage

What happens when we change state?

- How does it affect data on queues?
  - Retrospective?
  - Prospective?

- Single thread changes it immediately
- Multi thread *flush* the pipeline

What does flushing the pipeline do?

- Stalls the input
- Forces all data in queues to finish processing
- Change the state & returns

Avoid it except end of frame!

When Do Flushes Occur?

Yes!

- Matrix change
- Lighting change
- Texture change

No!

- Change attributes

- We could change parameters into attributes, but it would drive up bandwidth.

How do we prevent collisions on queues?

- Standard parallel problem: read/write conflict
- Avoid with a locking mechanism: E.g. std::lock_guard
- Lock the queue while reading or writing!

Queue Access

EXPENSIVE!

Lock input queue

Retrieve data from queue

Unlock input queue

Process data

Then lock, write & unlock output queue

Parallel Queue Access

- 32 processors
- Lock the queue once
- All 32 processors read one packet each
- Unlock the queue once
- SIMD/SMP parallelism
- Not thread-style parallelism
- And this is what *warps* do in hardware

Std::deque Problem

- std::deque automatically resizes
- Stalls memory management
- Standard solution: fixed (large) queue
- Input stalls if it's already full
- Until another thread starts clearing it

How do we know when we are done?

- There is usually a Flush() call at the end
- Or a SwapBuffers() which calls flush
- In Qt & similar, this is called automatically
- The simple solution doesn't need to: single threaded!

Assignment 1

- Implement the simple solution
- Single thread of control
- At end of Vertex(), call
  - TransformVertex()
  - RasterisePrimitive()
  - ProcessFragment()
- Framebuffer is a simple image

Framebuffer

- Real hardware can lock *per pixel*
- Or finesses it so it doesn't need to
- Doing this in software is expensive
- So we won't even try
- Just write directly to the framebuffer image
- And don't worry about locking

Approach

1. Take a sample programme

2. Make a list of all the functions called

3. Analyse variables

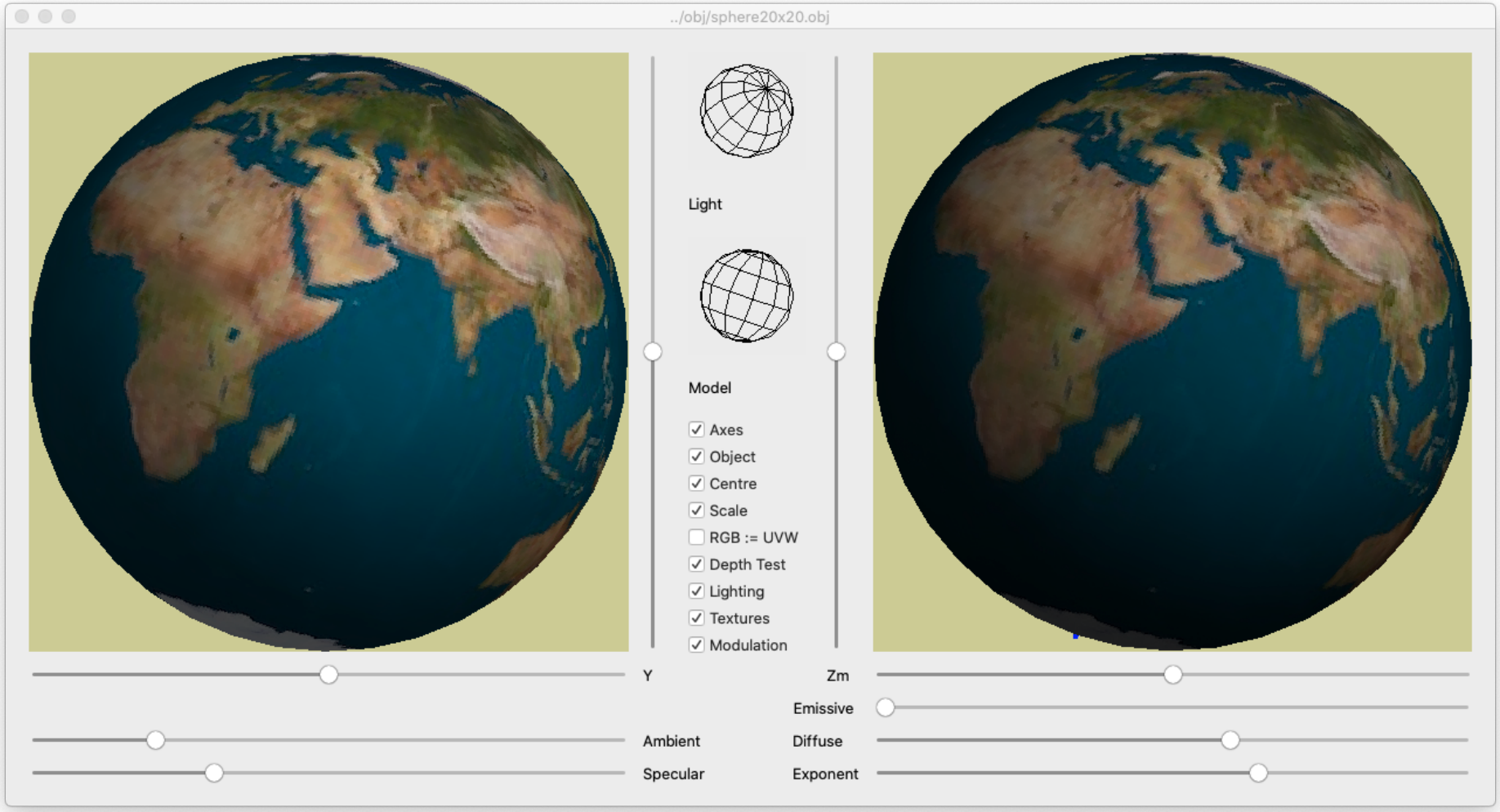
4. Write debug code

5. Design decision: threads & stages

6. Implementation Plan

7. Implementation

Sample Programme



OpenGL

FakeGL

Support Code

• Cartesian3	3D Cartesian coordinates
• Homogeneous4	Homogeneous coordinates
• Matrix4	Homogeneous matrices
• Quaternion	Quaternions
• RGBAValue	Colour values (bytes)
• RGBAImage	Simple image class (.ppm)
• TexturedObject	Object class (.obj/.ppm)

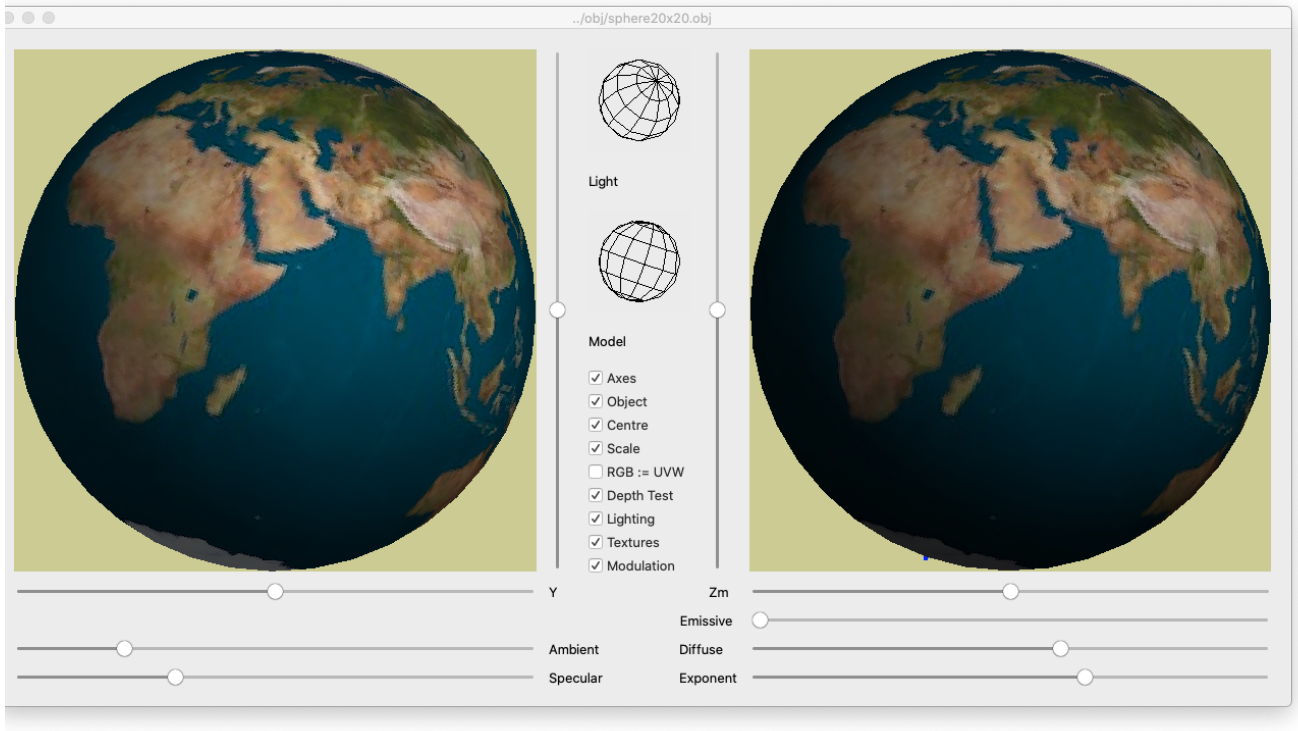
TexturedObject

- The core of our Model
- Texture & geometry specified separately
- Otherwise, reads in a subset of .obj file
- Has two nearly identical render routines
  - TexturedObject::Render()
  - TexturedObject::FakeGLRender()

UI Classes

- RenderParameters — Model class
- RenderWidget — View class
- RenderController — Controller class
- RenderWindow — Window class
- ArcBall/ArcBallWidget — ArcBall widget
- FakeGLRenderWidget — Your widget

FakeGLRenderWidget



- FakeGLRenderWidget
  - paintGL() paints image in widget
  - calls paintFakeGL() to paint image
  - paintFakeGL() has modified OpenGL calls
    - based on RenderWidget::paintGL()
    - calls TexturedObject::FakeGLRender()
  - changes here will be minimal

FakeGL.h/.cpp

- Based on a FakeGL context
  - a data structure storing all library state
  - three ancillary classes (later)
  - skeleton functions to implement
  - partial code for RasteriseTriangle()
- Your code goes here!

Recommended Approach

1. Decide on state variables & attributes
2. Implement stream output functions
3. Implement Clear()
4. Implement TransformVertex()
5. Implement RasterisePoint()
6. Implement ProcessFragment()
7. Implement PointSize()

Recommended Approach

8. Add matrix transformations
9. Implement Depth Test
10. Implement RasteriseLineSegment()
11. Add calls for RasteriseTriangle()
12. Implement Lighting in TransformVertex()
13. Implement Textures
14. Implement Phong shading

Deciding on state variables & attributes

- It's hard to know what you need, until you need it!
- Do this before starting each implementation step.
- Tips:
  - Look at how the methods are used on FakeGLRenderWidget/RenderWidget
  - Look at defined constants on FakeGL.h

Output stream functions

- After you have variables, implement printing them out.
- Useful for debug when you cannot see things on the screen
- Std::cout << fakeGL << std::endl; whenever you want to check the state.
- getchar() stalls so you can read output if you want

Clear()

- Sets every pixel in the frame buffer
- Just implement setting pixel to colour
- Tests whether we are showing frame buffer

TransformVertex()

- Reads vertex from vertexQueue
- Converts it to VCS
- Then computes lighting (if enabled)
- Then finishes conversion to DCS (why?)
- Sets position, colour, tex coords
- And adds vertex to rasterQueue
- Start with what you have (color) add the rest later.

RasterisePrimitive()

- Slightly tricky – depends on primitive type
  - points, lines or triangles
  - one subroutine for each
- Return false if queue is empty
  - Or if it doesn't have enough vertices
- For now, assume that calls are balanced

RasterisePoint()

- Only sets one pixel – hard to see
- But not impossible
- Render a small number of points
- And use the text output to debug it
- Hence the exercise the first week
- Then upgrade to add a set of fragments
  - in a box based on PointSize()

ProcessFragment()

- Simplest version always sets the pixel
- Based on the row & col retrieved from queue
- So implement that first
- Then add extra features one at a time

Depth test

- Now you are able to see points, and has gone through all the pipeline.
- Go through the implemented functions adding the variables/tests to implement the depth test.
- We will use an extra image as depth buffer
  - Store the depth in the alpha channel.

RasteriseLineSegment()

- Start by calling RasterisePoint() twice
- Then add line raster code:
  - Parametric
    - setting pixels directly
    - or calling RasterisePoint() repeatedly
  - Quad (needs triangle working first)

RasteriseTriangle()

- M.Eng. students did this last year
- M.Sc. students did not
- Don't want unfair advantage or boredom
- So I'm providing code for it
- But only for colour interpolation
- You will need to modify it a little bit for other functions

Lighting/textures

- Add necessary states/management/variables
- Modify implemented functions to add one feature at a time.
- If output starts to stall, use a simpler model.

More on lab later today!