08: Flushing & Buffers

Dr. Hamish Carr & Dr. Rafael Kuffner dos Anjos

Agenda

- Parallel processing
- Passing information to openGL
- Some optimizations

A Problem

- Suppose I want to render two surfaces
  - Each with a different texture
- I've avoided this in the first assignment
  - I only have one texture per object
  - And only object
  - And I process primitives immediately

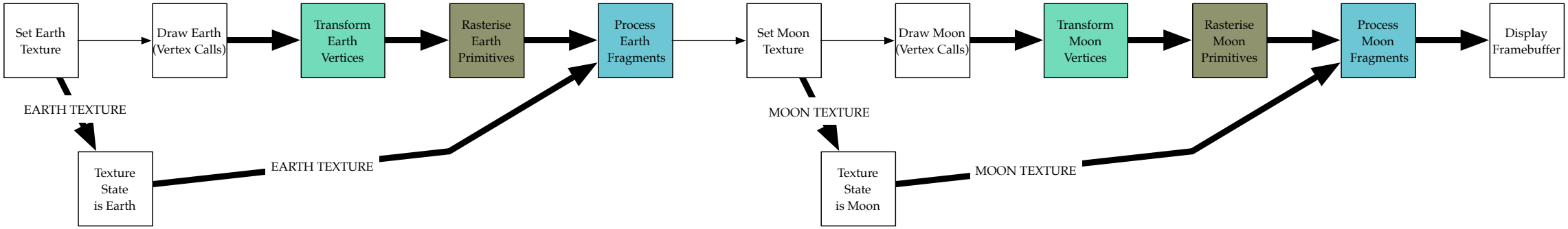
The Reality

- I add vertices to the queue
- A separate thread or threads processes them
- Another thread or many rasterises
- Another thread (or many) process fragments
- We have to deal with read/write order
- Assumption: we don't know when it happens
- What can go wrong?

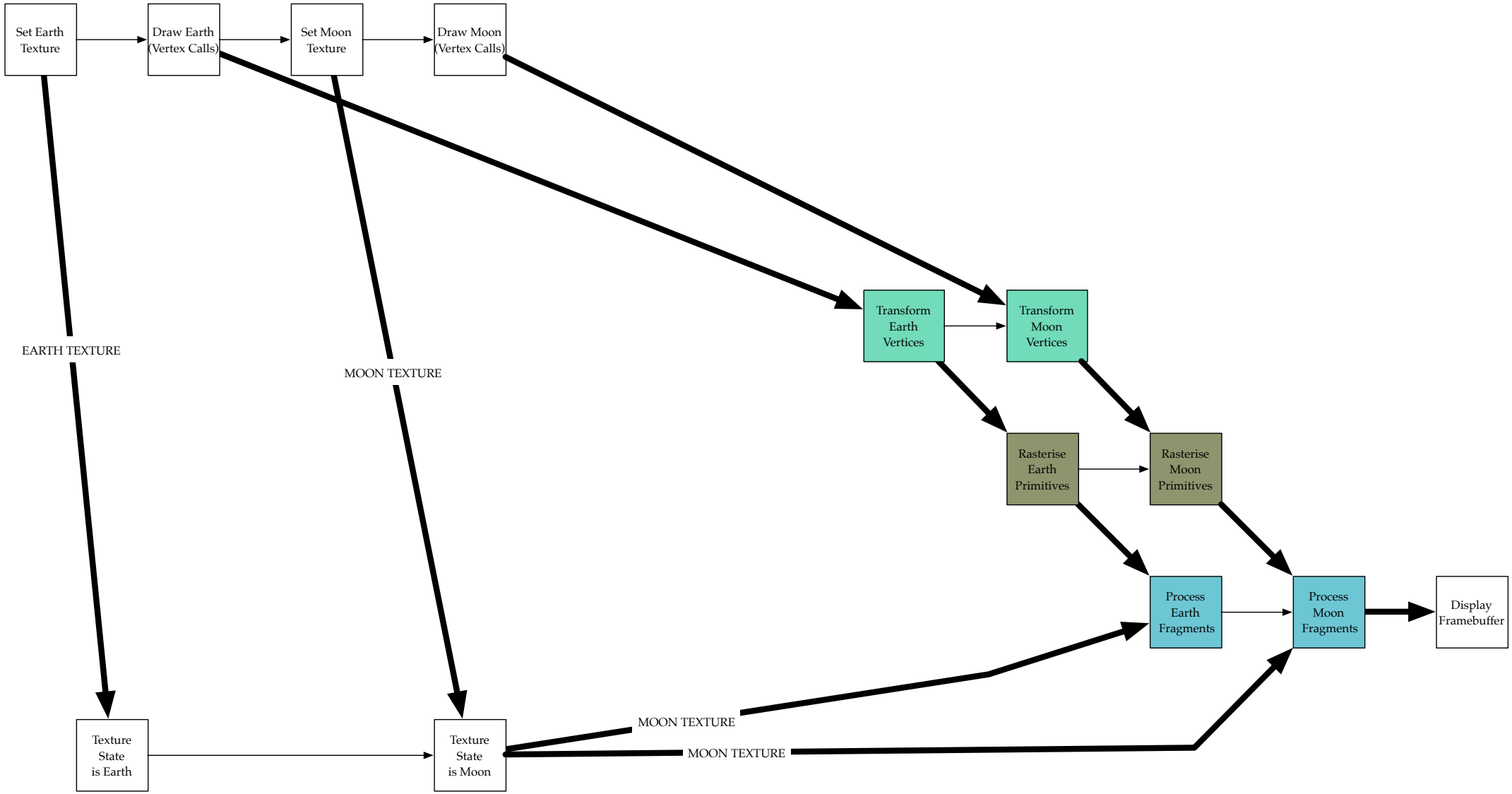
Out-of-Order Processing

- A general problem with parallelism
- One processor generates data
- Another processor consumes it
- If all data is passed explicitly, no problems
- But if any data is implicit, big problems
- For example, two textures

What we asked for (in serial)



What we got (in parallel)



Solutions

1

1. Keep everything serial (not happening)

2

2. Pass everything as part of data (expensive)

3

3. Keep everything in state (wrong output)

4

4. Flush the pipeline whenever state changes

- Any state change triggers a flush
- glEnable(), glDisable(), &c., &c., &c.

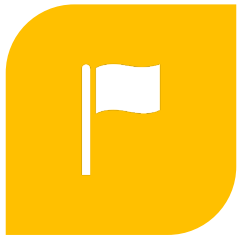
glFlush() Call

- Stalls or suspends the calling thread
- Invokes TransformVertex() until queue empty
- Then RasterisePrimitive() until queue empty
- Then ProcessFragment() until queue empty
- Releases the calling thread
- Net result: very expensive
- So minimise state changes in your pipeline

Calls that Flush

SET/MODIFY TEXTURES

SET/MODIFY MATRICES



CHANGES TO A RENDER FLAG

CHANGES IN LIGHTING

CHANGES TO RENDER ATTRIBUTES

Attributes

- Normally do not trigger flush
- Because they are *payload*
  - Data passed with the vertex / fragment
- The more attributes, the higher the cost

```
// class with vertex attributes
class vertexWithAttributes
    { // class vertexWithAttributes
    public:
    // Position in OCS
    Homogeneous4 position;
    // Colour
    RGBAValue colour;

    // you may need to add more state here

    }; // class vertexWithAttributes
```

Direct Mode

- So far, we have used *direct mode*
- We have called glVertex() for each vertex
- Which adds function call overhead
- And sends small packets down the video bus
- And last year, we computed vertex positions
  - For example, rendering a sphere
  - Every frame, we called trig functions

Improvements

Pre-compute vertex attributes (inc. position)

Treat vertex attributes as data (I/O)

Reuse vertices

Cache previous vertices

Group vertices together & process in bulk

Transfer attributes to texture

Tessellate higher order surfaces on the fly

1. Pre-computing Vertices

- Never, *ever* compute vertices every frame
- Unless you absolutely have to
- Compute once at beginning of run
  - Store them in memory
- If needed, update them on a separate thread
- Note: this combines with other strategies

2. Vertex Attributes are Data

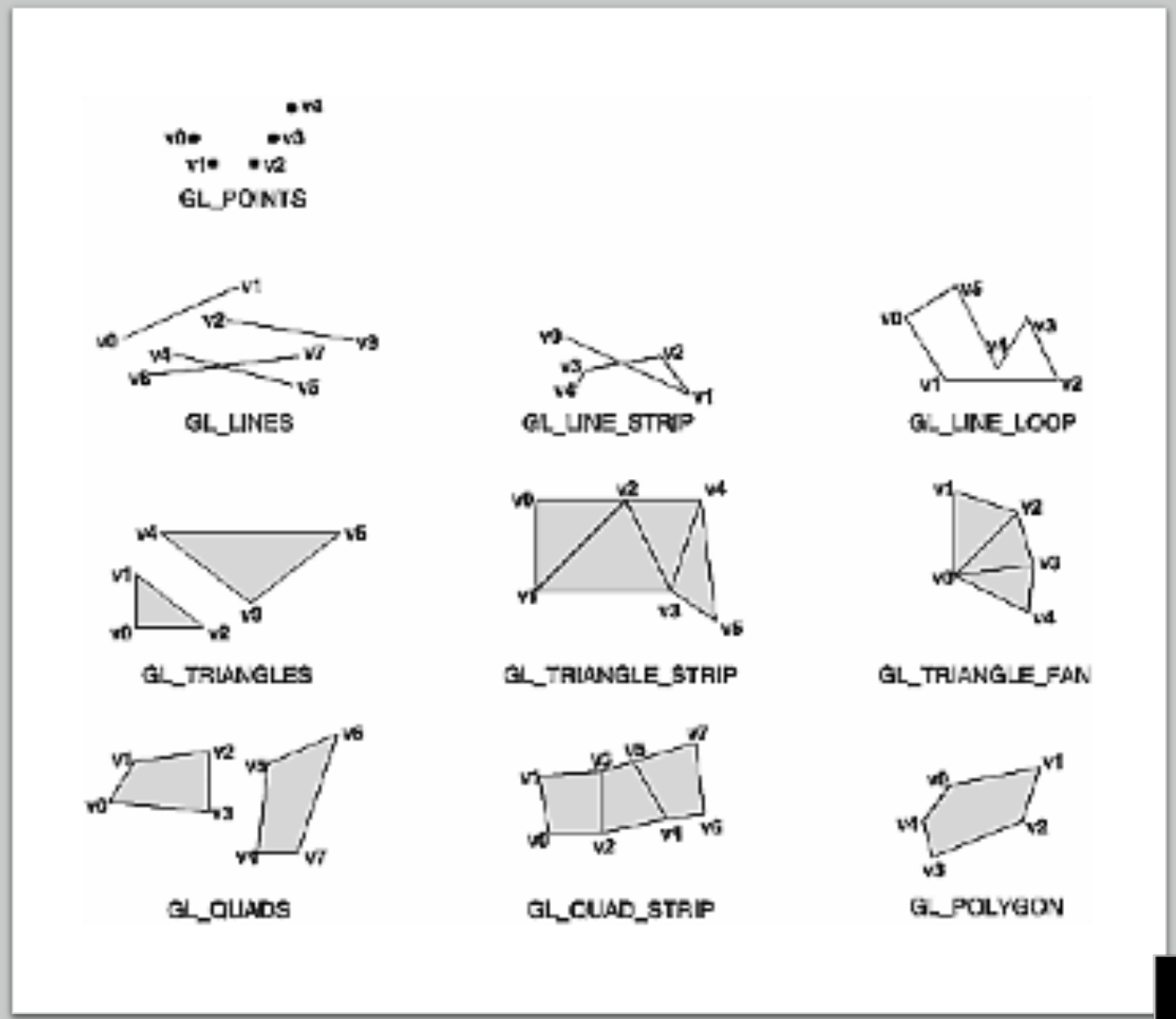
- If you can pre-compute, you can save to disk
- Store all your vertices on disk
- Read them in during load screen
- Then use them
- Paradigm: triangle soup
  - A disk file which lists vertices per face
- But on average, each vertex used 6 times

3. Re-use Vertices

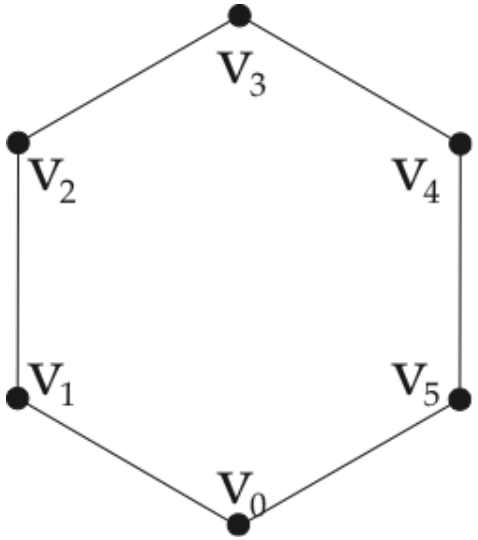
- Ideally, we transform each vertex once
- Store it in memory, then reuse it
- Three major variants:
  - library support for loops & strips
  - caching previous results
  - transfer to compute shader

(Old) Face Improvements

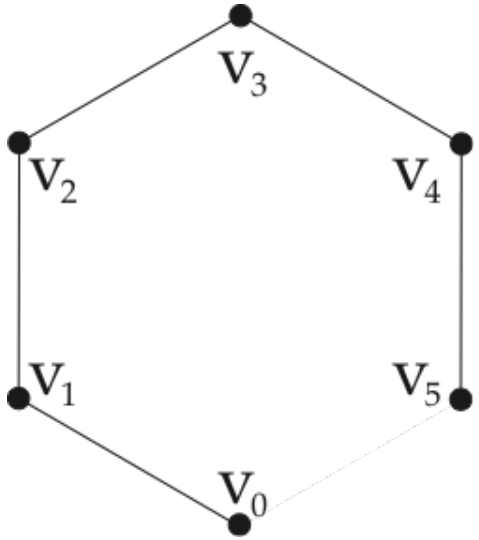
- Display lists – save bandwidth, cache on GPU
- Implicit vertex reuse
  - Line Strips & Loops
  - Triangle Fans & Strips
  - Quads, Quad Strips
  - Polygons



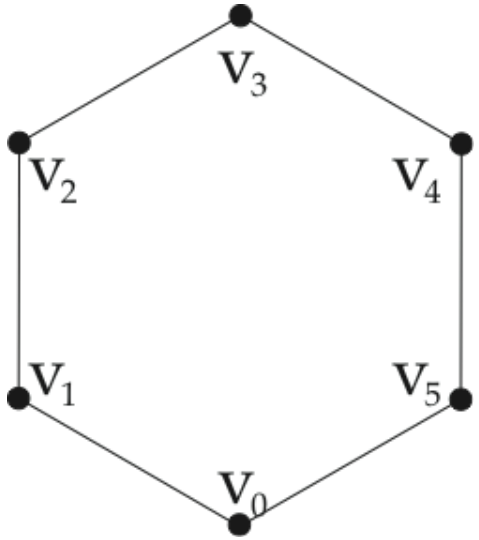
Lines, Strips & Loops



GL_LINES:
v0v1
v1v2
v2v3
v3v4
v4v5
v5v0
2 v/e

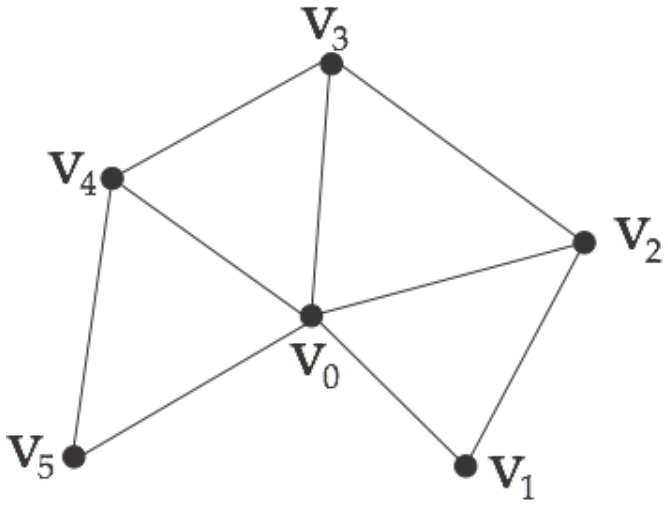


GL_LINE_STRIP:
v0v1
(v1)v2
(v2)v3
(v3)v4
(v4)v5
~ 1 v/e

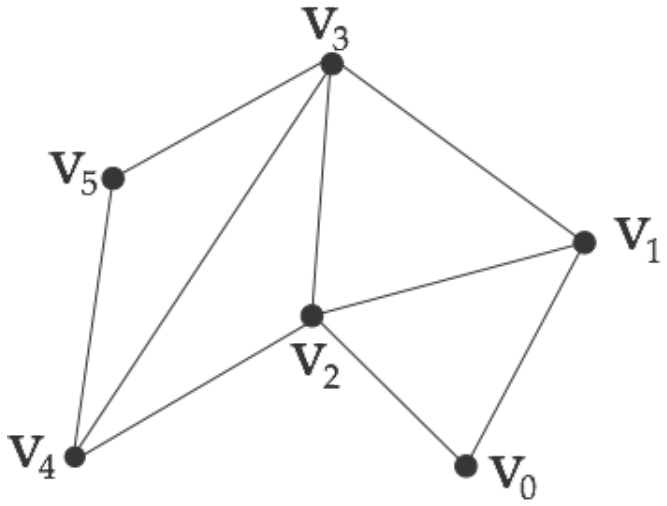


GL_LINE_LOOP:
v0v1
(v1)v2
(v2)v3
(v3)v4
(v4)v5
(v5)(v0)
~ 1 v/e

Triangle Fans/Strips



GL_TRIANGLE_FAN:
v0v1v2
(v0)(v2)v3
(v0)(v3)v4
(v0)(v4)v5
~ 1.16 v/e
(degree 6 vertex)



GL_TRIANGLE_STRIP:
v0v1v2
(v2)(v1)v3
(v2)(v3)v4
(v4)(v3)v5
~ 1 v/e
(for long strips)
alternate CW/CCW
NP-hard to find them
easier on regular grids

4. Caching Vertices

- Transform stage usually has a vertex TLB
  - Translation Lookaside Buffer
- Stores previously transformed vertices
- If no state has changed, uses previous result
- Clears the cache whenever the pipeline flushes
- Will never be entirely optimal
- But reasonably effective (and cheap)

5. Vertex Arrays

- Store all of the data in an array
- Loop through the array to use the vertices
- Can implement with:
  - array of classes
  - class of arrays
  - parallel arrays
  - interleaved arrays

Vertex Arrays

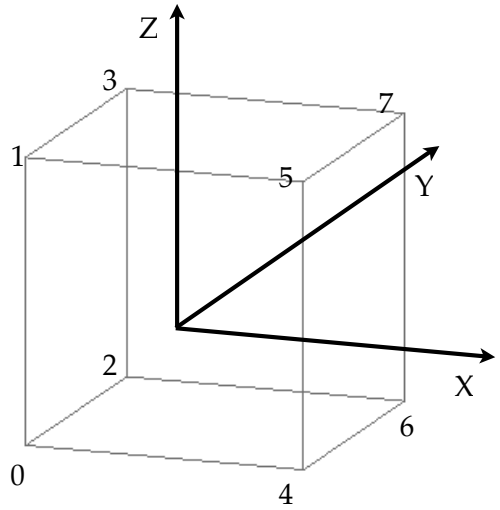
- Replace individual glVertex() calls - Pass *entire arrays* to library
- Push loop through vertices into library call

```
GLint vertices[8][3] = {  -1, -1, -1,   -1, -1,  1,   -1,  1, -1,  -1,  1,  1,
                           1, -1, -1,    1, -1,  1,    1,  1, -1,   1,  1,  1};
GLint normals[6][3] =  {  -1,  0,  0,    0, -1,  0,    0,  0, -1
                           1,  0,  0,    0,  1,  0,    0,  0,  1};
unsigned int triangles[36] = {  0, 1, 3,  0, 3, 2,  0, 1, 4,  1, 4, 5,
                                0, 2, 4,  2, 6, 4,  5, 4, 6,  5, 6, 7,
                                2, 3, 7,  2, 7, 6,  1, 5, 7,  1, 7, 3};

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_INT, 0, vertices);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(3, GL_INT, 0, normals);

// this runs it's own loop internally
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, indices);

// turn it off when we're done
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
```



GL Vertex Arrays

Notice, it just passes a pointer

So OpenGL has a CPU-resident loop

iterates through the array you passed it
makes the glVertex() calls for you
can optimise to reduce the number of calls

Essentially, this is what an .obj file is for

5a. Pretransformed Vertices

- Do all of the matrix transforms on CPU
- *Once* for each vertex
- Then use a vertex array
- Problem: transform stage *still* transforms them
- Unless you write a null transform stage
- But very useful for animation / skinning
  - Where vertices have weighted transforms

5b. Vertex Buffers

- An improvement on vertex arrays
- Instead of passing a pointer to the library, pass a pointer for bulk memory transfer directly to GPU
- Vertex array loop on GPU
  - Driver must have a standardized version
  - Adds some overhead

Working with Vertex Buffers

Where is it?

How do we allocate it?

How do we transfer data to it?

How do we connect it into the pipeline?

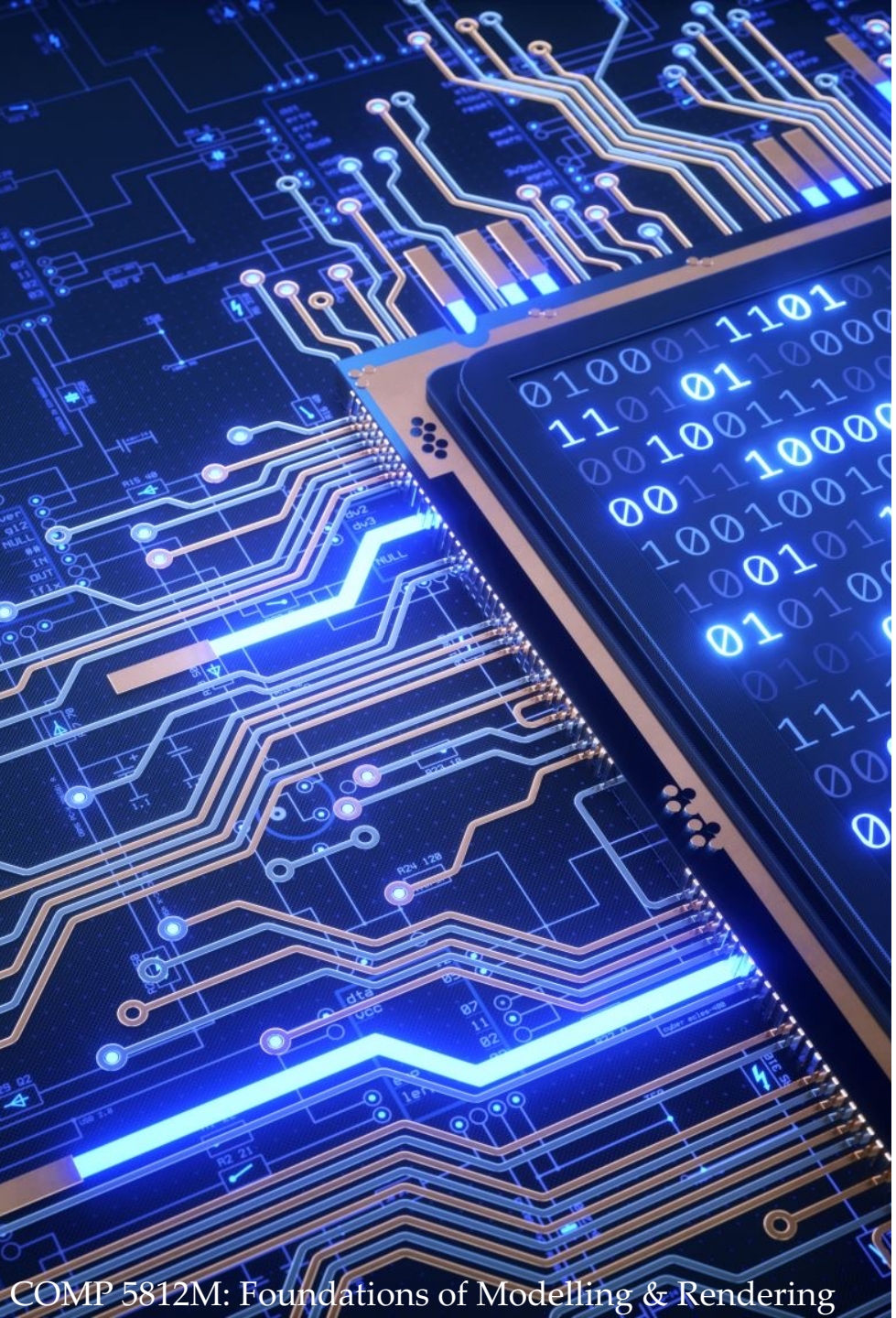
How does the shader access it?

Possible Locations

- A vertex buffer can be:
  - In main RAM
  - In shared (mapped) memory
  - In host-accessible VRAM
  - In GPU-only VRAM (fastest)
- The question is really how do we transfer it

Transferring the Data

- We want the data to end up in fast VRAM
- So we will have to copy it twice:
  - Once into CPU-accessible VRAM
    - Call this the *staging* buffer
  - Then into GPU-only fast VRAM
    - The actual vertex buffer



5c. Vertex Buffer Objects

- OpenGL name for one of these variants
  - A buffer on GPU + routines for transfer
  - Then bulk transfer vertices & attributes
  - I.e. you skip the library entirely
  - Just invoke a memory transfer
- Which *requires* O/S support

5d. Bulk Shader Transform

The pipeline assumes *streamed* vertices

Processes them one at a time

Instead, take a vertex buffer on the GPU

Transform each vertex (same as CPU)

Then feed into front end of pipeline

And set transform stage to null

But why do we then need a transform stage?

5e. The Future????

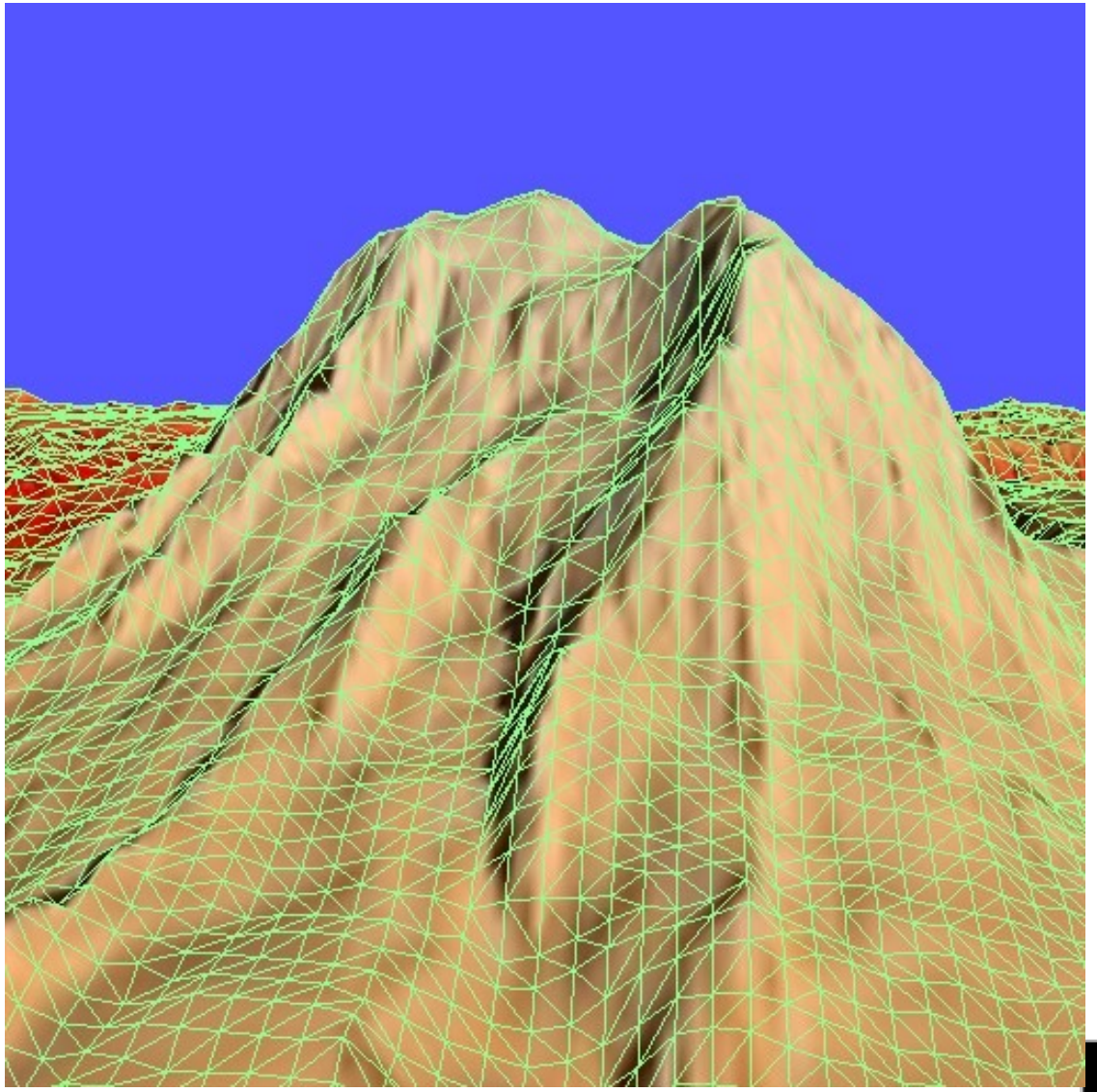
- Skip the entire pipeline
- Write your own render engine front to back
- Just embed it in a compute shader
- Emit the frame directly
- Requires access to hardware support
  - Eg frame buffer locking

6. Attribute Textures

- Most attributes of a surface are passed down (E.g. diffuse albedo (RGB))
- So why copy it from queue to queue?
- Store it in a texture and use only from the fragment shader
- Using the uv coordinates to look it up
- Minimises the pipeline cost

7. Surface Tessellation

- Objects designed with higher-order surfaces
- Then converted to triangles & stored on disk
- Why not just pass the higher order surface?
- Then tessellate at runtime?
- Generating normals, &c. on the fly
- Beginning to be very important
- As GPU horsepower has increased



Images by

- S2 Drew Beamer
- S8 Curology
- S11 Anne Nygård
- S12 Kaizen Nguyễn
- S15 Sigmund
- S19 Timo volz
- S26 Honey Yanibel Minaya Cruzs
- S27 Rose box
- S30 FLY:D
- On unsplash.com
- S32: visualization library v1.0.3