

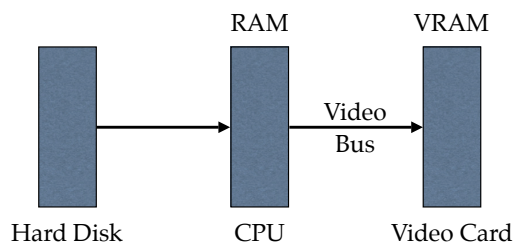
OpenGL Optimization

COMP 3003 Autumn 2005

Optimization

- Optimization is about *speeding it up*
- *Repeat* parts of rendering
- *Remove* redundant data
- *Discard* irrelevant data
- *Exploit* parallelism in hardware

Hardware Components



Disk vs. RAM Speed

- Serial ATA (fast disk):
 - 20 - 60 MB / s
 - Slower if it has to look for data
- 1.066 GHz Front Side Bus (FSB):
 - $266 \text{ MHz} \times 4 \times 4\text{B} = 3.2 \text{ GB} / \text{s}$

CPU Speed

- 3.5 GHz P4:
 - $3.5 \text{ GHz} \times 4\text{B} \times 2 \text{ (HT)} \times 3 \text{ (parallelism)}$
= 84 GB / s of data processed
- In practice, rarely manages this
 - but still *much* faster than RAM / disk
 - so we can use $\cos()$ & $\sin()$ for vertices

Video Bus Speed

- Theoretical maximum speeds:
 - PCI: $33 \text{ MHz} \times 4\text{B} = 133 \text{ MB} / \text{s}$
 - AGP: $66 \text{ MHz} \times 4\text{B} = 266 \text{ MB} / \text{s}$
 - AGP 2x: $66 \text{ MHz} \times 2 \times 4\text{B} = 533 \text{ MB} / \text{s}$
 - AGP 8x: $66 \text{ MHz} \times 8 \times 4\text{B} = 2.1 \text{ GB} / \text{s}$
 - PCI Express (16x): 4.2 GB / s

Video Bus Costs

- Texture: $128 \times 128 \times 32 \text{bpp} = 64 \text{ KB}$
- Geometry: each vertex has:
 - position (4f), normal (4f), colour (4f),
 - tex. coords (4f), materials (12f), &c.
 - 28 floats @ 4B each => 112B
- More than 300B / triangle (worst case)

How Many Triangles?

- At 30 fps (Hz) on a PCI bus:
 - $133 \text{ MB/s} \div 30 \text{ fps} \div 300 \text{ B/tri} = 13\text{K tris.}$
- At 60 fps (Hz) on an PCI-Express bus:
 - $4.2\text{GB/s} \div 60 \text{ fps} \div 300 \text{ B/tri} = 200\text{K tris.}$
- Not to mention textures & multiple passes
- But OpenGL state reduces triangle cost

Transform Rate

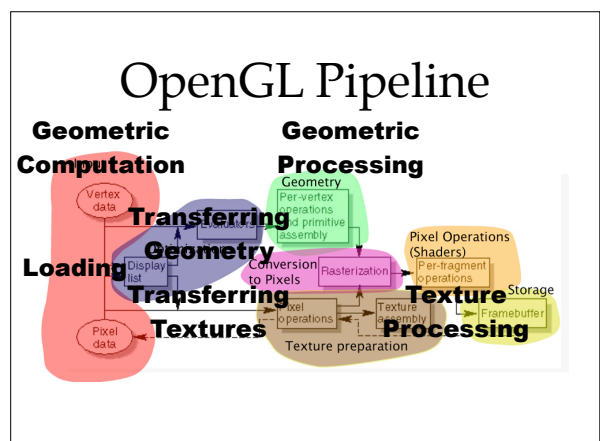
- How many vertices/sec. are transformed
 - GeForce 6800 Ultra DDL: 600 million
- Or 60 fps x 10M vertices
- Way faster than the bus can deliver
 - Unless we speed things up

Pixel Fill Rate

- How many pixels can be drawn / sec.
 - GeForce 6800 Ultra: 6.4G pixels
 - 16 pipelines at 400M pixels each
- At 60 fps, one rendering pass:
 - 100M pixels
 - $2560 \times 1600 \text{ screen} = 4\text{M pixels}$

Rendering Bottlenecks

- *Loading* from disk
- *Geometric computation* on the CPU
- *Memory usage* in RAM and VRAM
- *Transferring* data from RAM to VRAM
- *Geometric processing* on video card
- *Texture processing*



General Rules

- Optimize *early* in pipeline
- Minimize disk usage (*very* slow)
- Minimize video bus traffic
- *Cache* data close to where you use it
- *Add* more hardware (SLI, fragment processors, &c.)

Optimizing Loading

- Load textures in program initialization
 - Don't load textures you don't use
 - Don't reload textures
- Precompute geometry
 - e.g. positions of vertices on sphere
- Precompute animation data if possible

Optimizing Geometry

- *Cache* geometry on video card
- Use *repeated* geometric elements
- *Reuse* vertices where possible
 - Line strips & loops, triangle strips & fans
 - Use *vertex arrays* where possible
- *Substitute* textures for geometry
- *Discard* un-needed geometry

Inter-Frame Coherence

- Between two sequential frames
 - vertices are transformed geometrically
 - textures, colours, &c. don't change much
 - so why resend them over the video bus?
- We want to *cache* them in VRAM

Rendering Modes

- *Immediate* mode renders right away
 - processes commands & discards them
- *Retained* mode splits up processing
 - *definition*: transfers / caches commands
 - *invocation*: executes commands

Display Lists

- A *display list* is a set of OpenGL calls
 - *cached* on the video card
 - geometry, state changes, &c.
 - wheels on a train, parts of a body, entire sections of a world
 - identified by an *ID number*

Using Display Lists

- `glGenLists(nLists)` gets some new list IDs
 - returns first ID in consecutive range
- `glNewList(listID, GL_COMPILE)` starts storing OpenGL calls in a display list.
 - then make some OpenGL calls
- `glEndList()` stops storing OpenGL calls
- `glCallList(listID)` invokes the list

List Contents

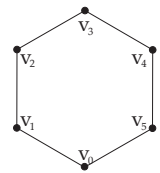
- Display lists can include:
 - `glCallList()`: other lists
 - `glVertex()`, `glNormal()`, `glTexCoord`, &c.
 - in fact, pretty much anything
 - with a few exceptions

List Limitations

- Display lists are stored in VRAM
 - vertex, normal, &c. are explicit
 - so 300B / triangle
- 100,000 triangles x 300B = 30MB
- Which is why we want lots of VRAM

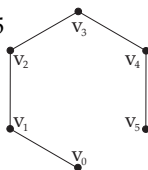
Reusing Vertices

- How many edges?
- How many vertices?
- But `GL_LINES` uses 12
 - `v0v1`, `v1v2`, `v2v3`, &c.
- We want to *reuse* vertices



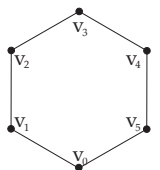
Line Strips

- Vertices in a strip are *repeated*:
 - `v0v1`, `v1v2`, `v2v3`, `v3v4`, `v4v5`
- Avoid redundant copies:
 - `glBegin(GL_LINE_STRIP)`
 - `v0`, `v1`, `v2`, `v3`, `v4`, `v5`
 - $n+1$ vertices for n lines



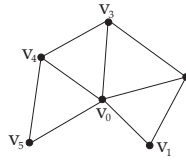
Line Loops

- Again, vertices are repeated
- `glBegin(GL_LINE_LOOP)`
 - same as line strip
 - last and first vertices paired as well
 - n vertices for n lines



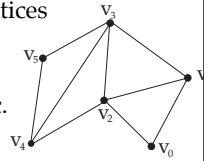
Triangle Fans

- `glBegin(GL_TRIANGLE_FAN)`
- All triangles share vertex v_0 :
 - $v_0v_1v_2, v_0v_2v_3, v_0v_3v_4, \&c.$
- Each is counterclockwise (CCW)
- $n+2$ vertices; n triangles

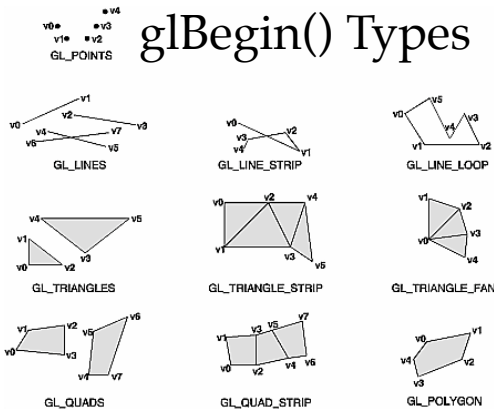


Triangle Strips

- `glBegin(GL_TRIANGLE_STRIP)`
- Adjacent triangles share 2 vertices
 - alternate CCW and CW
 - $v_0v_1v_2, v_1v_2v_3, v_2v_3v_4, \&c.$
- 4 triangles, 6 vertices
- $n+2$ vertices, n triangles



glBegin() Types



Finding Triangle Strips

- How long a triangle strip can we get?
 - Depends on the model
 - Easier on regularly-gridded models
- Computing strips is *difficult (NP-hard)*
 - So do it by hand (if at all)

Vertex Redundancy

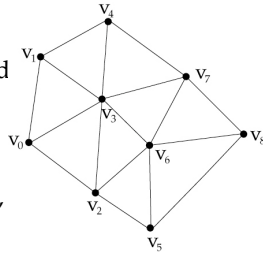
- In a *long* triangle strip:
 - Each triangle has 3 vertices
 - Each vertex is used 3 times
 - So count each as $1/3$ vertex
 - *Average* $3 \times (1/3) = 1$ vertex / triangle
- Is there any more redundancy?

Vertex Redundancy

- A *triangulated surface* has:
 - v vertices, e edges, and f triangular faces:
 - $v - e + f = 2 - g$ (genus: # of holes)
 - n_i vertices of *degree* i (with i edges):
 - $3n_3 + 2n_4 + 1n_5 + 0n_6 = 12 - g + n_7 + 2n_8 + \dots$
 - So most vertices are degree 6

Vertex Arrays

- Store vertices in *array*
- Send array to video card
- Define triangles with vertex indices:
 - 031, 032, 134, 263, 256, 367, 586, 687



Using Vertex Arrays

- `glEnableClientState(GL_VERTEX_ARRAY)`
- Use `glDisableClientState()` to turn off
- `glVertexPointer()` passes array to card
- `glArrayElement()` specifies vertex
- Requires precomputation of array
- Also possible for colour, normal, &c.

Optimizing Textures

- Use as few as possible
- Load them only as needed
- Cache them in VRAM
 - `glGenTextures()`, `glBindTextures()`
- Render objects with same texture together

Texture Objects

- A *texture object* is like a display list
- `glGenTextures()` gets IDs for texture objects
- `glBindTexture()` acts like `glNewList()`
 - specifies which texture object to use
 - and sets it as the active texture
- `glTexImage2D` sets the bound texture object

Multitexturing

- Complex visual effects use framebuffer
 - Often require multiple texture passes
- Newer hardware does them all at once
 - *multitexturing* (supported by OpenGL)
 - but still the same basic ideas

Shading Languages

- Multitexturing has become very complex
- Adding more features is expensive
- So *shading languages* developed:
 - allow *programmable* texture effects
 - applied by *fragment processor*

Scan-Line Interlace

- Cute trick used by nVidia
- Use two video cards
- Each renders alternate line of pixels
 - or alternate frames
- But geometry duplicated on both cards
 - works better for textures than geometry

Red Book Tips

- Provide unit length normals
- Use GL_FLAT if possible
- Pass vectors for precomputed values
- Pass scalars if computing on the fly
- Get rid of redundancy