The problem

- How to organise a large game world?
- The polygon-soup approach:
  - The world and everything in it consists of polygons
  - No further organisation
  - Process all polygons, render all polygons

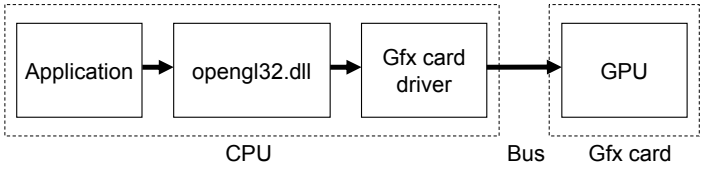
Polygon soup

```
struct Vertex
{
   float x, y, z;
};
struct Triangle
{
   Vertex v[3];
};
typedef vector<Triangle> World;
World world;

void RenderWorld(void)
{
   glBegin(GL_TRIANGLES);
   for(World::const_iterator wi = world.begin(); wi != world.end(); wi++)
   {
        glVertex3f(wi->v[0].x, wi->v[0].y, wi->v[0].z);
        glVertex3f(wi->v[1].x, wi->v[1].y, wi->v[1].z);
        glVertex3f(wi->v[2].x, wi->v[2].y, wi->v[2].z);
   }
   glEnd()
}
```


Polygon soup

- Polygon soup approach to rendering fails for any significant number of polygons
- Game runs on CPU, gfx are rendered by gfx card
- Everything that needs to be rendered has to be sent to the gfx card via the gfx driver



Polygon soup

- Three potential bottlenecks:
  1. CPU speed
  2. Bus speed
  3. Gfx card speed
     - Fillrate limit: how many pixels per second can be drawn
     - Triangle limit: how many triangles can be processed per second
     - Vertex limit: how many vertices can be processed per second
- The keys to faster rendering:
  - Reduce the amount of data that needs to be sent to the gfx card every frame
  - Reuse data that has already been sent

Objects

- Polygons can be grouped into objects
- Geometry of object often static
- Pre-process the object using display list
  - Assigns a unique ID to a list of OpenGL commands
  - Render by calling the list through the ID number
  - Driver can optimise the display list
  - If memory allows, driver may place display list on the gfx card

Objects (cont.)

```
struct Vertex
{
   float x, y, z;
};
struct Triangle
{
   Vertex v[3];
};
struct Object
{
   vector<Triangle> triangles;
   GLuint list;
};

vector<Object> world;
```


Objects (cont.)

```
void PreprocObject(Object& object)
{
   object.list = glGenLists(1);
   glNewList(object.list, GL_COMPILE);
   glBegin(GL_TRIANGLES);
   for(int t = 0; t < object.triangles.size(); t++)
   {
        glVertex3fv(&object.triangles[t].v[0]);
        glVertex3fv(&object.triangles[t].v[1]);
        glVertex3fv(&ojectt.triangles[t].v[1]);
   }
   glEnd();
   glEndList();
}


void RenderWorld(void)
{
   for(int o = 0; o < world.size(); o++)
        glCallList(world[o].list);
}
```

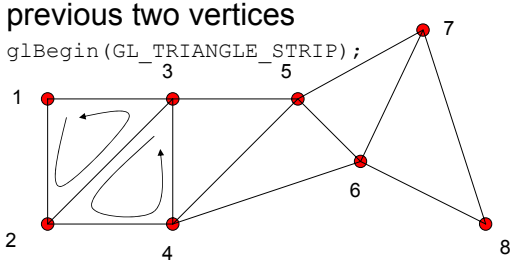

Objects (cont.)

- Vertices in an object often shared by several (3,4,5) polygons
- Using fewer vertices is good:
  - In OpenGL need to explicitly specify vertices and their properties, not polygons
  - GPU transforms vertices, not polygons, so the number of vertices is important
- So re-use vertices:
  - Strips and fans
  - Vertex arrays

Objects (cont.)

- Triangle strips
  - Each vertex creates a new triangle using the previous two vertices

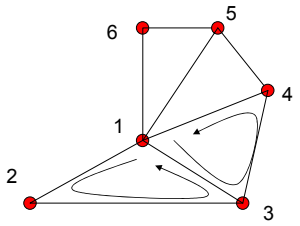
```
glBegin(GL_TRIANGLE_STRIP);
```



Objects (cont.)

- Triangle fans
  - Each vertex creates a new triangle using the first and previous vertex

```
glBegin(GL_TRIANGLE_FAN);
```



Objects (cont.)

- Vertex arrays
  - Store vertex positions, texture coords, etc. in arrays
  - Tell OpenGL to render triangles, triangle strips, quads, etc. using the info in the arrays
  - Fast, as no need to call `glVertex()` for each vertex

Objects (cont.)

- Creating vertex arrays:

```
float vertices[numverts][3];      // Vertex positions
float normals[numverts][3];       // Normals for each vertex
...               // Any other arrays, like texture coords

// Fill in the vertex arrays with data
...
```

- Tell OpenGL about the arrays:

```
glVertexPointer(Glint size, GLenum type, GLsizei stride, void* vertices);
glNormalPointer(GLenum type, GLsizei stride, void* normals);

glVertexPointer(3, GL_FLOAT, 0, vertices);
glNormalPointer(GL_FLOAT, 0, normals);
```

Similar for texture coordinates (`glTexCoordPointer()`) and colours (glColorPointer())

Objects (cont.)

- Enable the use of the arrays:

```
glEnableClientState(GLenum cap);

glEnableClientState(GL_NORMAL_ARRAY);
```

- Rendering using the arrays:

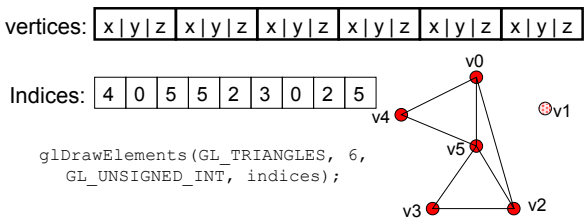
```
glDrawArrays(GLenum mode, GLint first, GLsizei count);

glDrawArrays(GL_TRIANGLE_STRIP, 0, numverts);
```


Objects (cont.)

- Rendering vertex arrays using indexing
  - Instead of using the vertices in array order, supply an array of indices into the vertex arrays

```
glDrawElements(GLenum mode, GLsizei count, GLenum type, void* indices);
```




```
glDrawElements(GL_TRIANGLES, 6,
  GL_UNSIGNED_INT, indices);
```


Objects (cont.)

```
float vertices[numverts][3];          // Vertex positions
float normals[numverts][3];           // Normals for each vertex
unsigned int triangles[numtris][3];  // Vertex indices making each triangle

// Fill in vertices, normals, triangles arrays
...

// Tell OpenGL about the arrays
glVertexPointer(3, GL_FLOAT, 0, vertices);
glNormalPointer(GL_FLOAT, 0, normals);

// Enable the use of the arrays
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);

...

// Use the enables arrays to draw triangles as many times as we like
glDrawElements(GL_TRIANGLES, numtris*3, GL_UNSIGNED_INT, triangles);
```


Objects (cont.)

- Vertex arrays can be put in display lists
- Arrays may be modified
- Performance could be improved if we could guarantee that arrays remain static
  - Done by using an extension: `glLockArraysEXT()`
- Driver/GPU can improve performance if the range of indices is known beforehand:
  - `glDrawRangeElements()` in OpenGL1.2

OpenGL extensions

- Not all functions are immediately available
  - Gfx-card specific extensions
  - opengl32.dll is OpenGL v.1.1
    - Current standard is (almost) 2.0
- To use those functions, must get function pointers from driver
  - glext.h header defines function prototypes
  - `wglGetProcAddress()` to get function pointers

OpenGL extensions (cont.)

```
#include <gl/glext.h>

PFNGLLOCKARRAYSEXTPROC glLockArraysEXT;

...

glLockArraysEXT =
   (PFNGLLOCKARRAYSEXTPROC)wglGetProcAddress("glLockArraysEXT");

...

if(glLockArraysEXT)
   glLockArraysEXT(0, numverts);
```


OpenGL extensions (cont.)

- Finding out what is supported:
  - OpenGL version string:

    ```
    GLubyte *version_str = glGetString(GL_VERSION);
    ```
  - Supported extensions:

    ```
    GLubyte *extensions_str = glGetString(GL_EXTENSIONS);
    ```

    Returns array of space-separated extension names
  - Especially useful for determining if a parameter value is supported