Introduction

• A bit of philosophy
• Review of the 3D pipeline
• Graphics architectures
• Rendering techniques
Graphics Programmers

• An effective graphics programmer...
  – ...knows “how to see”
    • deconstruct an image into first, second, third order effects
  – ...is familiar with human perception
    • psychology, biology
  – ...knows the medium
    • APIs, 3D hardware, content creation tools
  – ...knows the techniques
    • bump mapping, ray-casting, particle systems, etc.

• Synthesis of many disciplines
Game Graphics Programmers

- Create infrastructure to realise artistic vision of product
  - works closely with art director, artists
- Utilise the hardware and APIs of the target platform
- Integrate renderer with other game components
  - art pipeline, AI, front-end, etc.
- Achieve suitable performance levels
Realizing an Art Direction

• How would you render this?
Review of the 3D Pipeline

• Four stages:
  – Vertex transformation
    • Converts vertices to screen space triangles, generating any desired per-vertex components.
  – Polygon rasterization
    • Generate per pixel parameters
  – Fragment assembly
  – Frame buffer blending
    • Apply pixel colour to the frame buffer
Vertex Processing

- Vertices are comprised of one or more components
  - Position, colour, texture, normals, etc.
  - For programmable processing, a component may have arbitrary meaning.
  - Different processing steps can be performed on the components.
    - Components can be modified, or created on-the-fly:
      - Transform / project
      - Lighting (normals, lights -> colours)
      - Fog (depth, fog planes -> fog value)
      - Texture coordinate generation (eg. Sphere-map: normals -> texture co-ords)
      - Texture coordinate transformation (projection)
      - Interpolation, Skinning
Programmable vs. Fixed function

• Classically, vertex processing is fixed function
  – Select from a limited set of states to configure processing of components:
    • Fog on/off, light0-7, texgen none/sphere-map, etc.

• Modern hardware is programmable at the vertex level
  – Specialized instructions that are executed for each submitted vertex
    (vertex shader, PS2 VU1)
  – Tailored towards vector/colour operations
  – More or less general purpose depending on generation of hardware
    (conditionals, stack, loops, etc.)
  – C-like high-level language support
    • HLSL (DirectX), Cg (nVidia), GLSL (OpenGL 2.0)
OpenGL Shading Language

//
// dimple.vert: Vertex shader for bump mapping dimples (bumps)
//
// author: John Kessenich
//
// Copyright (c) 2002: 3Dlabs, Inc.
//

varying vec3 LightDir;
varying vec3 EyeDir;
varying vec3 Normal;

uniform vec3 LightPosition;
uniform float Scale;

void main(void)
{
    vec4 pos = gl_ModelViewMatrix * gl_Vertex;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec3 eyeDir = vec3(pos);
    gl_TexCoord[0] = gl_MultiTexCoord0;

    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
    vec3 t = normalize(cross(vec3(1.141, 2.78, 3.14), n));
    vec3 b = cross(n, t);

    vec3 v;
    v.x = dot(LightPosition, t);
    v.y = dot(LightPosition, b);
    v.z = dot(LightPosition, n);
    LightDir = normalize(v);

    v.x = dot(eyeDir, t);
    v.y = dot(eyeDir, b);
    v.z = dot(eyeDir, n);
    EyeDir = normalize(v);
}
Playstation 2 VU1

```
nop
nop[i]

nop
ilw.z vi01, GS_STATE_PACKET(vi00)z
nop
lq vf02,3(vi01) ; load fog colour

FOG_LOOP:

mulw ACC, _CTM3, vf00
maddaz ACC, _CTM2, vf01
madday ACC, _CTM1, vf01
maddx vf01, _CTM0, vf01

mulw.w vf01w,vf01w,FOGSCALE
addw.w vf01w,vf01w,FOGOFFSET
miniw.w vf01w, vf01w, i
maxw.w vf01w, vf01w, vf00x
ftoi0.w vf02w, vf01w

nop

iaddiu vi04,vi04,4 ; clamp to 255
iaddi vi05,vi05,-1 ; clamp to 0
sq vf02,1(vi06) ; store alpha into colour

nop

ibgtz vi05,FOG_LOOP

iaddiu vi06,vi06,3 ; incr pointer

; should adjust gftag to ignore ST!

nop

xgkick _GIFTAG

nop
```
Rasterization

- Given the three processed vertices
  - Generate vertex component values for each pixel inside the triangle
  - Interpolated w/ perspective correction
  - Position is no longer important, but depth (z) is stored
- Once depth is computed, it is tested against the z-buffer
  - If the test fails, no further processing is performed
  - Good thing too, as further processing can be very expensive with complex shaders
- Stencil test is also performed at this stage
Fragment Assembly

• By now, we have a depth value, some colours, some texture co-ordinates, and other values generated through interpolation of the vertex values
  – We want to convert these into a single colour and depth value (the fragment)

• Programmable or Fixed Function
  – Programmable is relatively new
  – Languages are not quite general-purpose yet
    • Significant performance implications running complex per-fragment programs
Fragment Assembly cont'd

- **Texture sampling**
  - Texture samples are fetched and combined with each other and other vertex-interpolated values
    - Either fixed function, or programmable
    - Many operations possible: add, subtract, scale, conditional, interpolate, dot-product
  - Number of texture samples per fragment depends on filtering mode (point, bilinear, trilinear), and number of active textures referenced by the primitive.
  - Result is a single colour value
Frame Buffer Blend

• Finally, we have a single colour (plus depth) for this pixel
  – This has to be combined with the colour in the frame buffer

• Operations
  – First the fragment undergoes an optional alpha-test
  – If the fragment fails the alpha test, further processing is not performed
  – Otherwise, the fragment colour is combined with the colour already in the frame buffer, and it is written back out.
    • src_colour*src_mode + dst_colour*dst_mode
  – If enabled, the z-buffer and stencil are also updated
Graphics Hardware Performance

- A few numbers to consider
  - Vertex processing rate
    - Depends on GPU clock rate + number of vertex ALUs available
    - And length of vertex program
  - Fragment processing rate
    - Depends on GPU clock rate + number of fragment ALUs available
    - And length of fragment program
  - Some hardware can share ALUs between vertex and fragment operations, other uses specific ALUs for each stage
  - Xbox360, Playstation 3 have 40+ ALUs on chip
    - Both type of processing is massively parallel
  - Pixel processing, texture fetch rate
    - Depends on GPU/VRAM bandwidth
Fill Rate (million pixels per second)

Triangle Rate (million triangles per second)
Renderer: Flow of Data

FIFO

Vertices & Indices  |  State Change

Host readback

GPU

Frame Buffer

Host (CPU)

GPU

- Colour read/write
- Z read/write
- Stencil read/write
- Texel read

Bus
Performance

• Graphics hardware is very fast and deeply pipelined
  – Likes to process vertices in gulps at a time
  – State switching between primitives slows things down
  – Host read-back stops everything dead in its tracks

• This puts a strong influence on graphics engine design
  – Favours brute-force, sent it up and let the hardware sort it out approaches
  – Don't chop the world up too much
  – Even if that means drawing lots off-screen
  – Batching by state
  – Avoid touching rendering data with the CPU
    • Compiled geometry, display lists
    • Can be stored in GPU-local memory
      – Avoids bus transfer
Performance cont'd

- Please don't do this!

```c
for( int i=0; i < nTris; i++ )
{
    glBegin(GL_TRIANGLES);
    glVertex(tri[i].p0);
    glVertex(tri[i].p1);
    glVertex(tri[i].p2);
    glEnd();
}
```

- Instead, use `glDrawArrays()`, `glDrawElements()`.
- Or at least take the `glBegin()`/`glEnd()` out of the loop.
Geometry Representation

• There are a few different ways to describe triangles to the graphics hardware.

• Interleaved vs. Separate component lists
  – Favour interleaved for best performance

• Triangles vs. Strips
  – Triangles: $n$ vertices defines $n / 3$ triangles
  – Strips: $n$ vertices defines $n-2$ triangles
    • 3x throughput increase assuming perfect strips
    • Need to stripify geometry off-line
    • Stitch together non-contiguous strips with “degenerate” vertices

• Indexed vs. Direct
  – Lower memory/bus requirements
  – Required to take advantage of GPU vertex caching
    • Especially important for programmable hardware
Scene Management

• Like the AI world representation, the renderer needs a representation of the world
  – Might be the same as the AI, but not necessarily
    • different operations are performed
    • different performance concerns

• Main operations
  – Visibility: what is visible from the current camera
    • Frustum culling
    • Occlusion
    • Level of detail
  – Rendering:
    • State batching
    • Translucency sorting
Culling

- Quickly reject items that are not within the viewing frustum
- Many techniques employed
  - Use simplified bounding volumes to test visibility against frustum planes
  - Spatial subdivision:
    - Grid, BSP-tree, kd-tree
  - Spatial clustering:
    - Sphere-tree, AABB-tree
  - Potentially Visible Set (PVS)
    - For a given viewing location, pre-compute conservative list of visible nodes
- Make sure tests take less CPU time than drawing the object!
Occlusion

- Quickly reject objects that are in the frustum, but can't be seen (covered up)
  - Analytically test conservative simplified representations of objects
  - Render representations into item-buffer (front-to-back), using z-buffer to resolve visibility
  - Potentially Visible Set
  - Portals
    - Anti-portals
      - Manually placed occlusion volumes
      - Defines an “anti-frustum”
  - Hierarchical coverage masks
- Some hardware support
Level of Detail

• Replace a complex representation of an object with a more simple representation when the complex detail isn't (as) visible
  – Object is far away
  – Object is small on the screen

• Many techniques
  – Simplify shaders at distance
  – Simplify animation at distance
  – Use lower polygon-count mesh
  – Mesh simplification (progressive meshes)
  – Fade-out
  – Terrain subdivision (ROAM, quadtree, row/column dropping)

• As with culling, make sure the extra work results in benefit.
Break
Lighting

• The most important aspect of the overall look
  – Difficult to get right

• Uses of lighting
  – can't see anything without lights
  – directing the viewer’s eye
  – creating depth
  – conveying time of day and season
  – enhancing mood, atmosphere, and drama
  – revealing character personality and situation
    • Sharon Calahan, Advanced Renderman

• Techniques borrowed from film, photography
  – We've got to roll our own methods
Lighting cont’d

• Lighting algorithms are concerned with
  – properties of the lights themselves
    • colour, intensity, shape
  – properties of the surfaces receiving the light
    • shininess, roughness, colour, transparency
  – things that affect the light en route to surfaces
    • Occlusion (shadows), scattering, refraction, filtering

• Physically-based or complex empirical models are extremely expensive
  – Can't model all the photons in real time
  – We have to fake it
  – Artists prefer it that way
Lighting cont’d

• Usually use multiple techniques in a renderer
  – Often world, dynamic objects, characters use different algorithms
• Ambient only
• Vertex-based approaches
  – Precomputed
  – Dynamic vertex lighting: directional/point/spot
• Texture-based approaches
  – Lightmaps, projected textures
Lighting cont’d

• Lighting systems often use separable algorithms so that effects can be layered
  – eg: colour * diffuse + specular + reflection
  – Layers can be combined using hardware combiners or with alpha blending and multiple rendering passes.

• Shadows require special consideration
  – Can be “baked in” for pre-computed lighting
  – Many algorithms to handle shadows when objects or light moves
    • shadow volumes, projected textures, projected geometry, ray casting, blobs
  – Even simple shadows add a lot to the look of a game
Texture-based lighting

- Per-pixel effects
  - if texture is high enough resolution
- Exploits high fill rate of modern hardware
- Arbitrary lighting equations
  - by casting them as 2D table lookups
- Light maps
- Projection
- Bump mapping
  - "normal maps" are stored in texture
  - Computed with a per-pixel light vector, texel dot product
Shadows

- Large effect on the lighting feel
- Gives hints of object height from the ground
- Many ad-hoc methods employed
Technique #1: Blobs

- Good first-order approximation
  - Much better than nothing
- Find the ground underneath the object
- Draw fuzzy, translucent blob texture of approximately the right shape
- Vary shadow size based on object height from ground
- Multiple blobs for articulated objects
  - Eg: one per leg, elbow
- Stretch and squash shadow based upon object speed
- Watch out for “blob surfing”
Technique #2: Projected Texture

• Render shadow caster from perspective of light source into temporary buffer
  – Often use lower-complexity model
• Project result onto shadow receiving surfaces into world
  – Or render as a decal
• Similar to blob, but with a nicer blob
• Extra rendering pass
• Have to watch for aliasing
Technique #3: Stencil Volumes

- Compute silhouette edges of geometry with respect to light.
- Extrude edges along light vector, creating shadow volume quads.
- Render all "shadow receivers", clear stencil buffer, mask colour writes:
  - Render front face volume, increment stencil on depth pass
  - Render back face volume, decrement stencil on depth pass
- Stencil buffer will contain 0 for areas of no shadow, >0 for areas of shadow:
  - Parity rule
- Simplest method to draw shadows: Draw full-screen "wash polygon" using stencil masking:
  - Doom 3 uses stencil buffer to mask shadow regions during lighting passes.
Stencil Volumes cont'd

- Works for arbitrary surfaces
- Handles self shadowing nicely
- Hard-edged shadows
- Silhouette edge calculation expensive
  - Can be precomputed if object<->light relationship doesn't change
- Large fill-rate requirements
  - Long skinny edges
- Standard approach breaks down if camera is inside volume
  - Parity rule broken
  - See “Carmack's reverse” for a solution
    - Robust Stencil Shadow Volumes, Mark J. Kilgard
Technique #4: Shadow Buffers

- Render scene into temporary buffer from light source (shadow buffer)
- For each pixel while rendering scene from camera position:
  - Back-project into “shadow space”
  - Texel lookup into shadow buffer
  - Modulate pixel with shadow colour
- General purpose, like stencil
- Expensive (multi-pass)
- Can suffer from severe aliasing
  - Shadow buffers usually have to be high-res
  - Pre, or post filter shadow to reduce aliasing (also gives soft shadows)
  - Very challenging to implement a general purpose system
2D Effects

- **Billboards**
  - View-vector aligned polygons
  - Useful for text, glows, particles
  - Placed in stacks, can be used as a cheap-ish volumetric effect
  - View-angle based fall-off effects to simulate lens flares
Frame Buffer Effects

• Utilize rendered data as a texture for subsequent operations
  – Render-to-texture, frame-buffer-as-texture

• Examples:
  – Motion blur
    • Render previous frame on top of current frame with alpha
  – Depth-of-field
    • Copy rendered frame to off-screen buffer, and blur
    • Using z-buffer as a mask, copy blurred frame to rendered frame
  – Refraction (Predator effect, heat shimmer)
    • Copy rendered frame to off-screen buffer
    • Render refracted object using frame-texture with perturbed texture co-ordinates
  – Feedback screen wipe
    • Draw copies of frame buffer on top of itself with additive blending, and rotation
Alpha Blending

• Z-buffers and alpha blending don't play together well
• Translucent objects should be drawn after opaque objects, and be sorted back-to-front
  – Otherwise viewing scene from certain angles will cause “portaling”
  – Additive objects don't need sorting step
• For “punch through” transparency, use alpha-test to reject 100% invisible pixels.
  – Not rendered to z-buffer means no artifacts
• New hardware can support fancier methods
  – Nvidia “depth peeling”
Wrap up

• Graphics programming is a synthesis of many disciplines
• Know the hardware
• Know the techniques
  – Many and varied
• Follow the research
• Look at other games
• Check out the demos on Nvidia, and ATI's sites
• Read Real Time Rendering