Computer Graphics

Accelerating 3D Graphics With Dedicated Hardware

Tobias Isenberg
Overview

• introduction to graphics hardware
• the graphics hardware (rendering) pipeline
• general purpose computation on the GPU (GPGPU)
What is graphics hardware?
What is graphics hardware?

- rendering traditionally implemented on CPU (C/C++/ASM)
- late 90’s: several companies (nVidia, ATI, 3Dfx) started releasing consumer hardware to remove rendering from CPU
- today: all desktops/laptops with dedicated graphics hardware
  - application logic still controlled by CPU
  - assets (3D meshes, texture maps, ...) uploaded to GPU at start up
  - CPU issues rendering commands to GPU
  - GPU performs rendering (transformations, lighting, etc)
  - results sent directly to the display
Why is graphics hardware effective?
Why is graphics hardware effective?

• example: AMD ATI Radeon HD 5870 (1600 cores)
• graphics engine: fixed-function hardware
• SIMD engines: single instruction, multiple data
  - i.e., lots of simple cores
  - massive parallel processing
  - perfect for graphics tasks
Why is graphics hardware effective?

CPU: multiple cores

GPU: hundreds to thousands of cores
Why is graphics hardware effective?

• 3D rendering can easily be parallelized:
  – meshes contain thousands of vertices & more; for each vertex we:
    • transform (object space $\rightarrow$ eye/camera space $\rightarrow$ screen space)
    • light (compute vectors, attenuation, etc.)
    • various other tasks …
  – rendered images have millions of pixels; for each pixel/fragment we:
    • interpolate coordinates
    • perform texture mapping
    • perform blending
    • compute illumination
    • various other tasks …
Why is graphics hardware effective?

- GPU throughput increasing faster than CPU throughput
Virtually all rendering requires GPUs
The computer graphics pipeline

- traditional pipeline can closely be mapped to the modern hardware/GPU pipeline
The graphics hardware pipeline

Index and vertex buffers (mesh data), Textures, etc

- Vertex Shader
- Tessellation
- Geometry Shader
- Rasterizer
- Fragment Shader
- Output Merger

= Configurable Stage
= Programmable Stage
= Memory Resource

= Pipeline Flow
= Memory Access

Note: terminology can vary between APIs; for example, OpenGL uses the term ‘fragment shader’, while Direct3D uses the term ‘pixel shader’
Controlling the pipeline (CPU)

• **graphics API:** programmer submits data/commands to GPU
  - OpenGL: open standard maintained by the Khronos group
  - OpenGL ES: cut-down version for use on embedded systems
  - Direct3D: developed by Microsoft for their systems
  - Vulkan: successor to OpenGL by the Khronos group

• **APIs are constantly evolving, e.g.:**
  - OpenGL 1.0 (1992): only configurable; some stages still missing
  - OpenGL 2.0 (2004): vertex and fragment stages programmable
  - OpenGL 3.0 (2008): added geometry shader as a programmable stage
  - OpenGL 4.0 (2010): added tessellation support as a programmable stage
Using an API

• before rendering commences, load relevant data onto the GPU
  – glGenBuffers(...), glBindBuffer(...), glBufferData(...), etc.
• also set up shaders for the programmable pipeline stages
  – glCreateShader(...), glShaderSource(...), glCompileShader(...), etc.
• once set up is complete, issue rendering commands
  – glDrawArrays, etc.
Vertex shader

Vertex Shader → Tessellation → Geometry Shader → Rasterizer → Fragment Shader → Output Merger

Index and vertex buffers (mesh data), Textures, etc
Vertex shader

- one of the first pipeline stages to become fully programmable (OpenGL 2.0)
- executed for each vertex in the input data
- most important role—apply transformations:
  - transform vertex into eye/camera space
  - project vertex into clip space
  - possibly lighting calculations (Gouraud shading)
Vertex shader: Example

• per-vertex diffuse lighting

```c
void main()
{
    // compute the diffuse light intensity
    vec3 normal = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightDir = normalize(vec3(gl_LightSource[0].position));
    float NdotL = max(dot(normal, lightDir), 0.0);
    vec4 diffuseLight = NdotL * gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;

    // assign the results to variables to be passed to the next stage
    gl_FrontColor = diffuseLight;
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```
Vertex shader: Example result
Vertex shader: Advanced uses

- particle systems
  - each particle modelled as single vertex
  - position and colour changed over time by the vertex shader
- animation
  - time passed to shader to animate the mesh
  - key-frame animation:
    - shader blends between predefined frames
  - skeletal animation:
    - each vertex is attached to a ‘bone’
    - CPU updates bone transformation
    - vertex shader applies this to each vertex
Vertex shader: Character animation
Tessellation shader

Index and vertex buffers (mesh data), Textures, etc
Tessellation shader

- recent addition to the hardware pipeline (OpenGL 4.0)
- used to increase the number of primitives via subdivision
- programmable, so various subdivision approaches possible
- effective when combined with displacement mapping
Geometry shader

Index and vertex buffers (mesh data), Textures, etc
Geometry shader

• relatively new addition to the pipeline (OpenGL 3.0)
• operates on primitives (e.g., lines and triangles)
  – input: usually the set of vertices the primitive consists of
  – shader has access to adjacency information
  – mesh processing algorithms such as smoothing and simplification
• modified geometry can also be saved to memory (*stream out*)
• multiple output primitives for each input primitive possible
Geometry shader: Duplication example

```c
void main(void)
{
  // output a copy tinted blue and raised up
  for(int i=0; i<gl_VerticesIn; i++) {
    gl_Position = gl_PositionIn[i] + vec4(0.0, 250.0, 0.0, 0.0);
    gl_FrontColor = gl_FrontColorIn[i] - vec4(0.3, 0.3, 0.0, 0.0);
    gl_TexCoord[0] = gl_TexCoordIn[i][0]; EmitVertex();
  }
  EndPrimitive();

  // output a copy tinted red and lowered down
  for(int i=0; i<gl_VerticesIn; i++) {
    gl_Position = gl_PositionIn[i] - vec4(0.0, 250.0, 0.0, 0.0);
    gl_FrontColor = gl_FrontColorIn[i] - vec4(0.0, 0.3, 0.3, 0.0);
    gl_TexCoord[0] = gl_TexCoordIn[i][0]; EmitVertex();
  }
  EndPrimitive();
}
```
Geometry shader: Example result
Geometry shader: Advanced uses

- procedural geometry
  - can also generate primitives procedurally
  - e.g., metaballs; mathematical surface which can be evaluated on the GPU

- particle systems
  - particles usually drawn as quad (requires four vertices)
  - send a single point, shader expands it into a quad

- shadow volume extrusion
  - extrude object boundary in shader, reduces CPU load
  - boundary used to determine what is in shadow
Index and vertex buffers (mesh data), Textures, etc
Rasterizer

- converts primitives into fragments
  - performs culling and clipping of primitives
  - generates fragments from primitives
  - property interpolation (color, texture coordinates, etc.)

- not programmable, but configurable:
  - backface culling
  - anti-aliasing
  - depth biasing
Fragment shader

Index and vertex buffers (mesh data), Textures, etc
Fragment shader

- also one of the oldest programmable stages (OpenGL 2.0)
- calculates fragment colour based on interpolated vertex values, texture data, and user supplied variables.
- fragment: ‘candidate pixel’
  - may end up as a pixel in final image
  - may get overwritten, combined with other fragments, etc.
- common uses: per-pixel lighting, texture application
Fragment shader: Example

• texturing and fog

```c
void main(void)
{
    // sample the texture at the position given by texcoords
    vec4 textureSample = texture2D(checkerboard, gl_TexCoord[0].st);

    // compute some depth-based fog
    const float fogDensity = 0.0015;
    float depth = gl_FragCoord.z / gl_FragCoord.w;
    float fogFactor = 1.0 - (depth * fogDensity);

    // compute the output color
    gl_FragColor = gl_Color * textureSample * fogFactor;
}
```
Fragment shader: Example result
Fragment shader: Advanced uses

• procedural textures
  – compute texture based on an algorithm
  – Perlin noise, Voronoi noise, fractals, etc.

• reflection
  – environment map, stored in cubemap texture
  – applied to object, accounting for the view direction

• normal (bump) mapping
  – add extra surface detail to a model
  – adjust the surface normal based on bump map
Index and vertex buffers (mesh data), Textures, etc
Output merger

- combines fragment shader output with any existing contents of the render target
- key roles:
  - depth testing (z-buffer)
  - blending: combine fragment color with pre-existing pixel in the render target (transparency, lighting, etc.)
- configurable, but not programmable
Performance considerations

• pipeline approach:
  – minimize state changes to avoid flushes
  – balance workload across stages

• slow memory: values can also be computed (to avoid look-up)

• quality/performance trade-off by moving operations between vertex and fragment shader (e.g., lighting)

• vertices often shared by multiple triangles
  – GPU implements a caching mechanism to avoid reprocessing
  – order triangles so that those sharing a vertex are rendered consecutively
The future?

• real-time photorealism still not achieved
  – “Requires roughly 2000x today’s best GPU hardware” (Tim Sweeney in 2012)

• continuing increase in the power of GPUs
  – more pixels (screens: UHD/4K, 8K, …)
  – more detail (i.e., triangles)
  – more processing (animation, physics simulation, AI, …)
  – increasing programmability and flexibility

• different direction:
  real-time raytracing? voxel/point-based graphics?

• increasing use of graphics hardware for non-graphics tasks
General purpose computing on the GPU

- increasing GPU programmability: application beyond graphics
- most effective for problems with a high degree of parallelism
  - define a *kernel* and apply it to many pieces of data simultaneously
- example applications:
  - image processing: blurring/sharpening, segmentation, feature detection, etc.
  - physics simulation: fluid simulation, rigid bodies, cloth, etc.
  - non-shading rendering: raytracing, radiosity
General purpose computing on the GPU

• CPU is still in overall control (like when rendering)
  – typically only small part of an application moved to the GPU
• several APIs for GPGPU computing
  – OpenCL (Khronos group)
  – CUDA (nVidia)
  – DirectCompute (Microsoft)
• several generalized concepts:
  – use of general arrays instead of operation on textures/render targets
  – more flexible memory access
• additional concepts:
  – support for synchronisation between processing cores
GPGPU—Simple example: Adding arrays

CPU

// allocate some arrays
const int num = 10;
float *a = new float[num];
float *b = new float[num];
float *c = new float[num];

// fill 'a' and 'b' with some data
for(int i = 0; i < num; i++) {
    a[i] = b[i] = 1.0f * i;
}

// compute 'c' as the sum of 'a' and 'b'
for(int i = 0; i < num; i++) {
    c[i] = a[i] + b[i];
}

// print the result
for(int i = 0; i < num; i++) {
    printf("c[%d] = %f\n", i, c[i]);
}

parallelized using GPGPU (OpenCL)

__kernel void AddArrays(__global float* a, __global float* b, __global float* c)
{
    // determine which element of the array we are working on
    unsigned int i = get_global_id(0);

    // perform the addition
    c[i] = a[i] + b[i];
}

CPU setup code (some details omitted)

// create an OpenCL kernel from the kernel source
cl_kernel kernel = clCreateKernel(program, "AddArrays", &err);

// upload the data to the GPU
cl_mem cl_a = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
    sizeof(float) * num, a, &err);
cl_mem cl_b = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
    sizeof(float) * num, b, &err);
cl_mem cl_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(float) * num, NULL, &err);

// execute the kernel
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, workGroupSize,
    NULL, 0, NULL, &event);
GPGPU & Combination with rendering

• general computation and rendering possible in one program
  – e.g., simulation on the GPU, render results also with the GPU
  – CPU can be removed from the process almost entirely!

• example: fluid simulation on the GPU
  – fluid treated as large number of particles in an array
  – kernel applied to each array element to update positions
  – particles rendered as spheres, adding smoothing and refraction
GPGPU & Combination with rendering