Extending the Graphics Pipeline with Adaptive, Multi-Rate Shading

Yong He

Joint work with Yan Gu and Kayvon Fatahalian
CMU 15-869 Fall 2013
GPUs must render images significantly more efficiently to meet the demands of future displays.

- GPU: 4.0 MPixels, GPU: ~2 Watts
- GPU: 3.7 MPixels, GPU: 200 Watts

100×
GPUs must render images significantly more efficiently to meet the demands of future displays.

Dell 27 inch 5K display
5120 * 2880
14.7 MPixels
Today’s Topic

My recent research that evolves the GPU’s graphics pipeline to render high resolution images more efficiently.

• Background
  • What is a graphics pipeline and how it is used in games

• New Pipeline Architecture
  • That works more efficiently

• Adapting Applications
  • How to implement several popular graphics effects on the new pipeline

• Shading Language
  • Language and compiler support for the new pipeline

• Evaluation
Background: how do GPUs render an image?
Representing a 3D scene as Triangles

Triangle #1:
(0.0, 0.0, 0.0)
(1.0, 0.0, 0.0)
(0.0, 1.0, 0.0)

Triangle #2:
...
Albedo Texture and Texture Coordinates

Triangle #1:
(0.0, 0.0, 0.0)
(0.0, 0.0)
(1.0, 0.0, 0.0)
(1.0, 0.0)
(0.0, 1.0, 0.0)
(0.0, 1.0)

Triangle #2:
...
What does a graphics pipeline do?

Vertex Data  
(3D Position, Texture Coordinate...)

+  

Textures  
(and other resources...)

→

Rendered Image
Step 1: Project 3D Coordinates into 2D screen

Triangle #1:
(0.0, 0.0, 0.0)
(1.0, 0.0, 0.0)
(0.0, 1.0, 0.0)

Triangle #2:
...

Vertex Shader
Step 2: Rasterize Triangle into Pixels

Covered pixels are called **fragments**.
Step 3: Compute Color for Each Fragment (Fragment Shading)
The Graphics Pipeline

Vertex0: \((x_0, y_0, z_0)\)
Vertex1: \((x_1, y_1, z_1)\)  (Triangle Coordinates In 3D space)
Vertex2: \((x_2, y_2, z_2)\)

- **Vertex Buffer**
- **Vertex Shading**  (Run Vertex Shader)
- **Transformed Vertex Buffer**
- **Rasterize**
- **Fragment Buffer**
- **Fragment Shading**  (Run **Fragment Shader**)
- **Blend Ops**
Fragment shaders compute many shading effects

- Texturing (albedo texture)
- Lighting
- Shadows
- Reflection
- Atmosphere Scattering (fog)
- ...

**Fragment shaders are usually expensive in games.**
Let’s focus on fragment shading!

Vertex0: (x0,y0,z0)
Vertex1: (x1,y1,z1)  (Triangle Coordinates In 3D space)
Vertex2: (x2,y2,z2)

Vertex Shading  (Run Vertex Shader)

Rasterize

Fragment Shading  (Run Fragment Shader)

Blend Ops
Fragment Shader Example

```cpp
struct FragmentInput
{
  float3 world_space_position;
  float3 surface_normal;
  float3 texture_coord;
};

float4 FragmentShader(FragmentInput input)
{
  float3 lightPosition = float3(0.0, 100.0, 0.0);
  float3 albedo = texture("brick.bmp", input.texture_coord);
  float lighting = ComputeLighting(lightPosition, input.world_space_position, input.surface_normal);
  return albedo * lighting;
}
```
How to speed up fragment shading?
A typical fragment shader computes many effects

\[ \text{Surface Albedo} \times \text{Lighting & Shadows} + f(\text{Other Artistic Effects...}) \]
Not all effects need to be computed once per pixel.

Lighting and Shadows:

- Mostly low-frequency
- Expensive to compute (lots of arithmetic instructions!)
Some still need to be computed once per pixel, but they are cheap!

Surface albedo:
- High frequency
- Cheap to compute (only 1 texture fetch!)
It is sufficient to sample many expensive shading effects less than once per pixel...
It is sufficient to sample many expensive shading effects less than once per pixel...
It is sufficient to sample many expensive shading effects less than once per pixel...

Lighting & Shadows

Adaptive Sampling

Albedo
Adaptive, Multi-Rate Shading

Shadowed Lighting Shading Result Albedo
Adaptive, Multi-Rate Shading
3x fewer shading instructions

Per-Pixel Shading (Reference)
Difference Image (magnified 10 times)
Difference Image (magnified 100 times)
Contribution: robustly extending the real-time graphics pipeline for adaptive, multi-rate shading
Contribution: robustly extending the real-time graphics pipeline for adaptive, multi-rate shading

- GPU pipeline architecture extensions for multi-rate shading
- Scheduling adaptive workloads efficiently
- Design of robust adaptive shaders
- New adaptive, multi-rate shading language
Pipeline Architecture
Multi-Rate Pipeline Architecture

Rasterize → Coarse Shading → gm → Fine Shading → Blend Ops
Multi-Rate Pipeline Architecture

Rasterize $\rightarrow$ Coarse Shading $\rightarrow$ Fine Shading $\rightarrow$ Blend Ops

Fine Fragments
Multi-Rate Pipeline Architecture

Rasterize → **Coarse Shading** → **Fine Shading** → Blend Ops

Coarse Fragments

Fine Fragments
Multi-Rate Pipeline Architecture

Rasterize → Coarse Shading → Resample Bilinear → Fine Shading → Blend Ops

Coarse Fragment → Fine Fragments (w/ interpolated inputs) → Fragment output colors
coarse_lighting = /* compute lighting... */

albedo = texture(uv);
frag_color = albedo * coarse_lighting;
The Multi-Rate Shading Process

1. Rasterizer emits coarse fragments
2. Coarse shader executed on coarse samples
The Multi-Rate Shading Process

1. Rasterizer emits coarse fragments
2. Coarse shader executed on coarse samples
The Multi-Rate Shading Process

1. Rasterizer emits coarse fragments
2. Coarse shader executed on coarse samples
The Multi-Rate Shading Process

1. Rasterizer emits coarse fragments
2. Coarse shader executed on coarse samples
3. Coarse shading results resampled to each pixel location
The Multi-Rate Shading Process

1. Rasterizer emits coarse fragments
2. Coarse shader executed on coarse samples
3. Coarse shading results resampled to each pixel location
4. Execute fine shader, which combines the coarsely shaded effects with other fine effects
Coarse Fragment Shader

coarse_lighting = /* compute lighting... */

coarse_shadow = /* compute shadow term... */

if (block overlaps penumbra)
  shadow_refine = true;

Fine Fragment Shader

albedo = texture(uv);

if (shadow_refine)
  shadow = /* compute shadow term... */
else
  shadow = coarse_shadow;

frag_color = albedo
  * coarse_lighting
  * shadow;
Three inputs to fine fragment shading:
1. Triangle attributes
2. Coarse shading results
3. Refinement flags
Pipeline Details
Fragment shader are executed on groups of 2x2 pixels for screen-space derivatives

- Many effects (e.g. texture lookup) require screen-space derivative of a certain value

- Screen-space derivatives can be easily obtained if neighboring fragments are grouped into 2x2 blocks and executed in SIMD fashion

- In traditional GPU pipeline, rasterizer outputs coverage results at 2x2 pixel granularity
Coarse shader is executed on 2x2 coarse samples, covering 4x4 pixels

- For multi-rate pipelines, rasterizer outputs coverage results at 4x4 (or larger) pixel block granularity, instead of 2x2 pixel blocks.
Scheduling Adaptive Workload

Coverage Masks (per 4x4 pixel block)

Coarse Shading

Coarse Shading Samples

Refinement Flags

Quad Fragments

Generate Fine Shading Work

Rasterize

Repack (optionally sort by refinement flags)

Fine Shading SIMD Packets
Designing Adaptive Shaders
Compute lighting effect: diffuse + specular

- Lighting is computed as sum of diffuse term and specular term

\[
\text{Lighting} = \text{Diffuse (lower frequency)} + \text{Specular (higher frequency)}
\]
**Diffuse surfaces: check difference of coarse results**

**Coarse Shader:**

```cpp
coarse_diffuse = /* compute diffuse lighting... */
if (difference(coarse_diffuse) > THRESHOLD)
    refine_flag = 1;
else
    refine_flag = 0;
```
Specular surfaces: highlight bounds check

curvature only refinement

curvature + highlight bounds refinement
Shadows: determine if surface block lies in penumbra
Rendering shadows using a shadow map
Shadow Map Example
Shadows: determine if surface block lies in penumbra
Shadow edges correspond to large discontinuities in shadow edge map

Check:
1. Point position correspond to an edge pixel in shadow map
2. Point lies in the farther side of the shadow map edge
Preprocess shadow maps to identify penumbra regions

“Shadow Edge” Map

[Hasselgren 2007]
Multi-Rate Shading Language
One function per stage programming

struct Vertex_In {
    float3 vertex, normal;
    float3 view, light_dir;
    float2 uv;
}

struct Coarse_Out {
    float diffuse, specular;
    bool diffuse_flag : SV_REFINE_FLAG_0;
    bool specular_flag : SV_REFINE_FLAG_1;
}

Coarse_Out coarse_shader(Vertex_In in) {
    Coarse_Out rs;
    rs.diffuse_flag = rs.specular_flag = false;
    float nDotL = dot(in.normal, in.light_dir);
    if (fwidth(nDotL) > DIFFUSE_THRESHOLD)
        rs.diffuse_flag = true;
    else
        rs.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
        rs.specular = spec_lighting(in, rs.specular_flag);
    else
        rs.specular_flag = true;
    return rs;
}

float4 fine_shader(Vertex_In vin, Coarse_Out cin) {
    if (cin.diffuse_flag) {
        float nDotL = dot(vin.normal, vin.light_dir);
        cin.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    }
    if (cin.specular_flag) {
        bool tmp_flag;
        // unused flag, but needed by call below
        cin.specular = spec_lighting(vin, tmp_flag);
    }
    float lighting = cin.diffuse + cin.specular;
    float4 albedo = texture(texAlbedo, uv);
    return albedo * lighting;
}

float4 spec_lighting(Vertex_In in, out int flag) {
    // compute specular lighting
    // optionally set flag if refinement is needed.
    ...
    return specular;
}
One function per stage programming

```cpp
struct Vertex_In {
    float3 vertex, normal;
    float3 view, light_dir;
    float2 uv;
}
struct Coarse_Out {
    float diffuse, specular;
    bool diffuse_flag : SV_REFINE_FLAG_0;
    bool specular_flag : SV_REFINE_FLAG_1;
}

Coarse_Out coarse_shader(Vertex_In in) {
    Coarse_Out rs;
    rs.diffuse_flag = rs.specular_flag = false;
    float nDotL = dot(in.normal, in.light_dir);
    if (fwidth(nDotL) > DIFFUSE_THRESHOLD)
        rs.diffuse_flag = true;
    else
        rs.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
        rs.specular = spec_lighting(in, rs.specular_flag);
    else
        rs.specular_flag = true;
    return rs;
}

float4 fine_shader(Vertex_In vin, Coarse_Out cin) {
    if (cin.diffuse_flag) {
        float nDotL = dot(vin.normal, vin.light_dir);
        cin.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    }
    if (cin.specular_flag) {
        bool tmp_flag;
        // unused flag, but needed by call below
        cin.specular = spec_lighting(vin, tmp_flag);
    }
    float lighting = cin.diffuse + cin.specular;
    float4 albedo = texture(texAlbedo, uv);
    return albedo * lighting;
}

float4 spec_lighting(Vertex_In in, out int flag) {
    // compute specular lighting
    // optionally set flag if refinement is needed.
    ...
    return specular;
}
```
struct Vertex_In {
  float3 vertex, normal;
  float3 view, light_dir;
  float2 uv;
}

struct Coarse_Out {
  float diffuse, specular;
  bool diffuse_flag : SV_REFINE_FLAG_0;
  bool specular_flag : SV_REFINE_FLAG_1;
}

Coarse_Out coarse_shader(Vertex_In in) {
  Coarse_Out rs;
  rs.diffuse_flag = rs.specular_flag = false;
  float nDotL = dot(in.normal, in.light_dir);
  if (fwidth(nDotL) > DIFFUSE_THRESHOLD)
    rs.diffuse_flag = true;
  else
    rs.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
  if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
    // unused flag, but needed by call below
    rs.specular = spec_lighting(in);
  else
    rs.specular_flag = true;
  return rs;
}

float4 fine_shader(Vertex_In vin, Coarse_Out cin) {
  if (cin.diffuse_flag) {
    float nDotL = dot(vin.normal, vin.light_dir);
    cin.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
  }
  if (cin.specular_flag) {
    bool tmp_flag;
    // compute specular lighting
    cin.specular = spec_lighting(vin, tmp_flag);
  }
  float lighting = cin.diffuse + cin.specular;
  float4 albedo = texture(texAlbedo, uv);
  return albedo * lighting;
}

float4 spec_lighting(Vertex_In in, out int flag) {
  // optionally set flag if refinement is needed.
  // compute specular lighting
  ... return specular;
struct Vertex_In {
    float3 vertex, normal;
    float3 view, light_dir;
    float2 uv;
}

struct Coarse_Out {
    float diffuse, specular;
    bool diffuse_flag : SV_REFINE_FLAG_0;
    bool specular_flag : SV_REFINE_FLAG_1;
}

Coarse_Out coarse_shader(Vertex_In in) {
    Coarse_Out rs;
    rs.diffuse
    float nDotL = dot(in.normal, in.light_dir);
    if (fwidth(nDotL) > DIFFUSE_THRESHOLD)
        rs.diffuse_flag = true;
    else
        rs.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    
    if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
        rs.specular_flag = true;
    else
        rs.specular_flag = true;
    return rs;
}

float4 fine_shader(Vertex_In vin, Coarse_Out cin) {
    if (cin.diffuse_flag) {
        float nDotL = dot(vin.normal, vin.light_dir);
        cin.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    }
    if (cin.specular_flag) {
        bool tmp_flag;
        // unused flag, but needed by call below
        cin.specular = spec_lighting(vin, tmp_flag);
    }
    float4 lighting = cin.diffuse + cin.specular;
    float4 albedo = texture(texAlbedo, uv);
    return albedo * lighting;
}

float4 spec_lighting(Vertex_In in, out int flag) {
    // compute specular lighting
    // optionally set flag if refinement is needed.

    return specular;
}
Declarative shading language for adaptive, multi-rate effects

in float3 normal, view, light_dir;
in float2 uv;
uniform sampler2d texAlbedo;

coarse effect float diffuse {
    diffuse = clamp(dot(light_dir, normal), 0, 1) * Kd + Ka;
    if (fwidth(diffuse) > DIFFUSE_THRESHOLD)
        refine;
}

coarse effect float specular {
    if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
        refine;
    specular = compute_specular(view, light_dir, normal);
}

fine effect float4 albedo = texture(texAlbedo, uv);
fine out float4 color = albedo * (diffuse + specular);
Compiling adaptive effect DAGs

- Diffuse (adaptive)
- Specular (adaptive)
- Albedo (fine)

Color (output)

Coarse Shader Code

Refinement Flags

coarse-to-fine interface

Fine Shader Code
Compilation of a simple multi-rate shader

coarse effect float diffuse {
    diffuse = /* Compute Diffuse Lighting */;
    if (/* Adaptive Predicate */) {
        refine;
    }
}

texture

fine effect float4 albedo = texture(texAlbedo, uv);
fine out float4 color = albedo * diffuse;

dothing
Compilation of a simple multi-rate shader

coarse effect float diffuse {
    diffuse = /* Compute Diffuse Lighting */;
    if (/* Adaptive Predicate */) 
        refine;
}

fine effect float4 albedo = texture(texAlbedo, uv);
fine out float4 color = albedo * diffuse;

struct Coarse_Output {
    float diffuse;
    int refine_flag : 1;
}

Coarse_Output coarse_shader() {
    Coarse_Output result;
    result.diffuse = /* Compute Diffuse Lighting */;
    result.refine_flag = /* Adaptive Predicate */;
    return result;
}

Fine_Output fine_shader(Coarse_Output coarse_in) {
    Float_Output result; 
    result.diffuse = coarse_in.diffuse;
    if (coarse_in.refine_flag) 
        diffuse = /* Compute Diffuse Lighting */
    float4 albedo = texture(texAlbedo, uv);
    result.color = albedo * diffuse;
    return result;
}
Minimize coarse-to-fine interface

- Maintaining a coarse-to-fine value is not free
  - Coarse-to-fine values are stored in scarce cache memory
  - Values need to be resampled to fine pixel locations
Minimize coarse-to-fine interface

Coarse

B (coarse)

C (fine)

Fine

Interface: B
Minimize coarse-to-fine interface

- A (adaptive)
- B (adaptive)
- C (coarse)
- D (fine)
Minimize coarse-to-fine interface

A (adaptive)  B (adaptive)

C (coarse)

D (fine)

Interface: A or B or C (1 slot)

1. None of A and B needs refinement: pass C
2. Both A and B need refinement: pass nothing
3. Only A needs refinement: pass B
4. Only B needs refinement: pass A
Generating minimal coarse-to-fine interface

1. Determine which effects could potentially be passed
   • Need to pass effect A from coarse to fine iff there is at least one effect B s.t. B depend on A and there exists a possibility when B is computed in fine stage while A is computed in coarse stage.

2. Register-allocate these effects to interface slots
   • Two effects interfere when there exists a possibility that both effects need to be passed from coarse to fine.
Evaluation
Evaluation Setup

• Multi-rate pipeline configurations:
  • COARSE2X2: coarse shading = once per 2x2 pixels
  • COARSE4X4: coarse shading = once per 4x4 pixels
  • DYNAMIC:
    Attempt once per 4x4 pixels shading,
    then attempt once per 2x2 pixels if required

• All performance results include costs of SIMD divergence that result from adaptive logic in shaders
Reduction in Number of Shader Instructions (relative to per-pixel shading)

Rendered at 2560x1440

2x2: coarse2x2  4x4: coarse4x4  DYN: dynamic
2-5x Reduction on Low Geometric Complexity Scenes

Rendered at 2560x1440
Benefits of multi-rate shading decrease for small triangle scenes

Hairball

2x2: coarse2x2

4x4: coarse4x4

DYN: dynamic

Relative instruction count (vs. reference GPU shading)
Scheduling Adaptive Workload

Coverage Masks
(Per 4x4 pixel block)

Coarse Shading Samples

Refinement Flags

Quad Fragments

Generate Fine Shading Work

Repack (optionally sort by refinement flags)
SIMD Execution Efficiency

No sorting

Sort fine fragments based on refinement flags
Detailed shadows in San-Miguel trigger extra refinements.
Performance Stability Under Animation

Relative instruction counts (vs. reference GPU shading)

- Sponza
- Station

Frames: 240

- COARSE2x2
- COARSE4x4
Cost benefits are stable across frames

Relative instruction counts (vs. reference GPU shading)

Sponza

Station
Summary
Adaptive Multi-Rate Shading

• Preserves image quality while reducing cost up to a factor of five

• Benefits increase with increasing display resolutions

• Mechanism applicable to other contexts: foveated rendering, motion blur
Adaptive Multi-Rate Shading

• Preserves image quality while reducing cost up to a factor of five

• Benefits increase with increasing display resolutions

• Mechanism applicable to other contexts: foveated rendering, motion blur

• Important limitation:
  • Detailed/bumpy surfaces require fine sampling
  • Coarse fragment shading does not benefit dense triangle meshes
Related work

• Multi-resolution rendering
  • Render each shading effect into a stand-alone image with different resolutions, then up-scale each effect image to full-screen and combine
  • Require drawing entire scene multiple times
  • Increased memory accesses
  • Aliasing, and no adaptivity support

• Coarse Pixel Shading [Vaidyanathan et al.]
  • Introduces the coarse shading stage, eliminating the need for drawing scene multiple times
  • Does not support adaptivity
Meeting the demand of future advanced displays

High DPI
Mobile

8000 x 8000 @ 90Hz VR