OpenGL ES
Shading and Advanced Rendering

Luc Semeria and Michael Swift
iPhone GPU Software

Agenda

OpenGL ES 2.0

- Recap of graphics pipeline
- Basics of programmable shading

Real-time rendering techniques

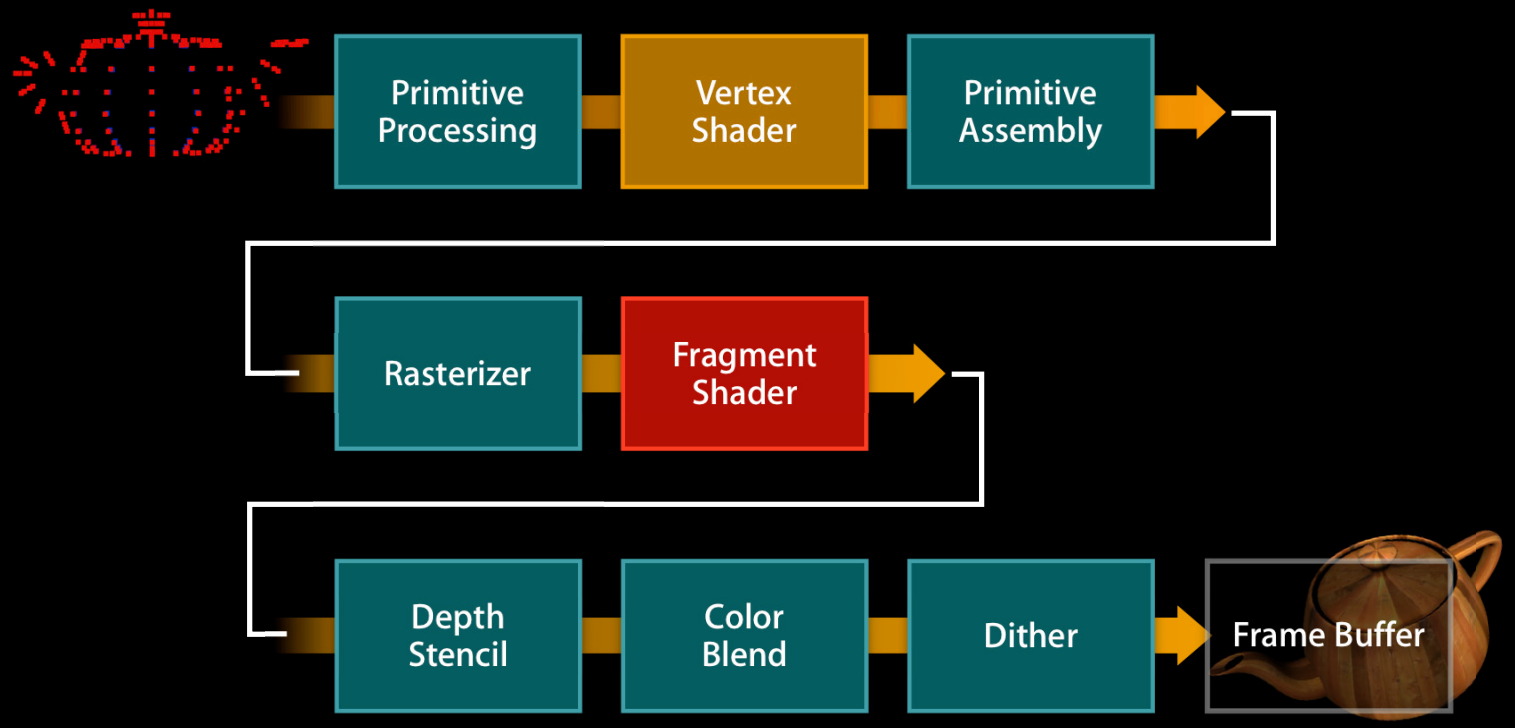
- Skinning
- Lighting
- Shadowing

Agenda

- Recap of graphics pipeline
- Basics of programmable shading

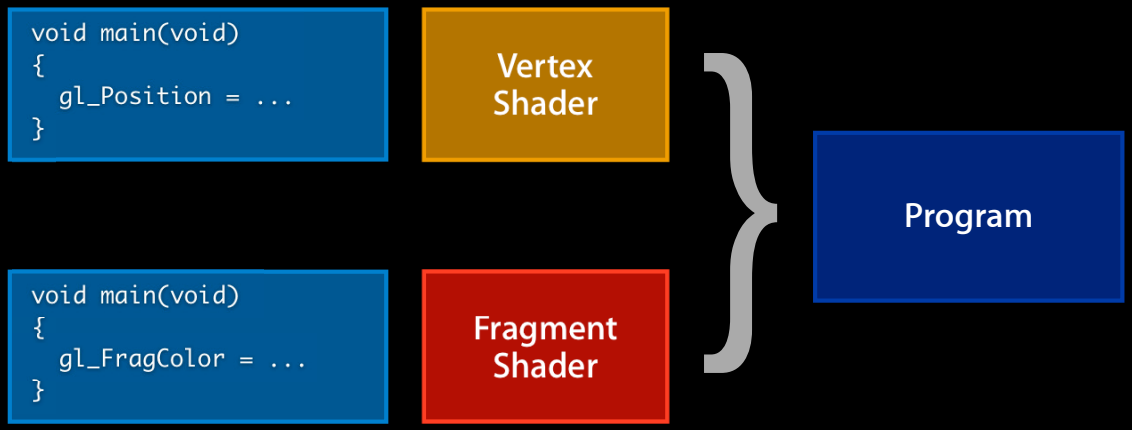
OpenGL ES 2.0

Programmable graphics pipeline



OpenGL ES 2.0

Vertex and Fragment Shaders



Programmable Graphics Pipeline

Powerful

- Tangent space bump map
- Cubic environment mapping
- Refraction
- Better image processing

Flexible

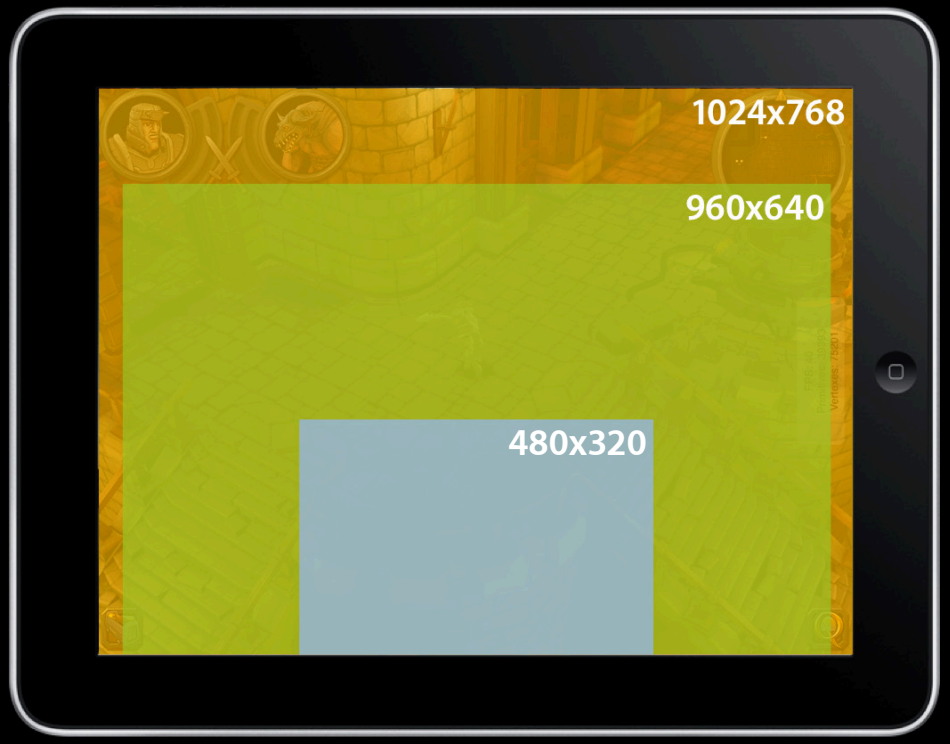
- Algorithm selection
- Performance tuning

Programmable Graphics Pipeline

Support

- iPhone 3GS
- iPod touch 3rd gen.
- iPad
- iPhone 4

PowerVR SGX GPU

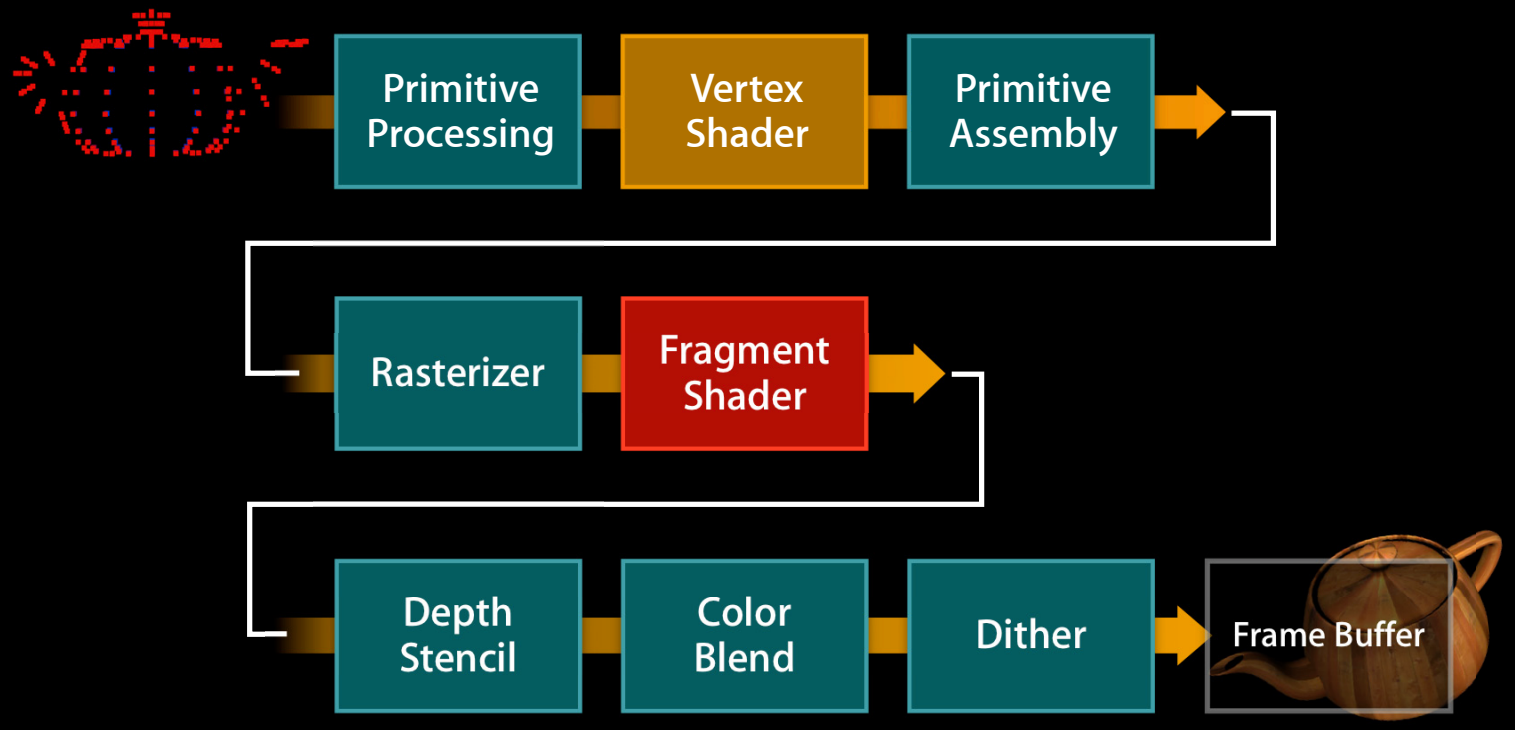


Agenda

- Recap of graphics pipeline
- Basics of programmable shading

OpenGL ES 2.0

Programmable graphics pipeline



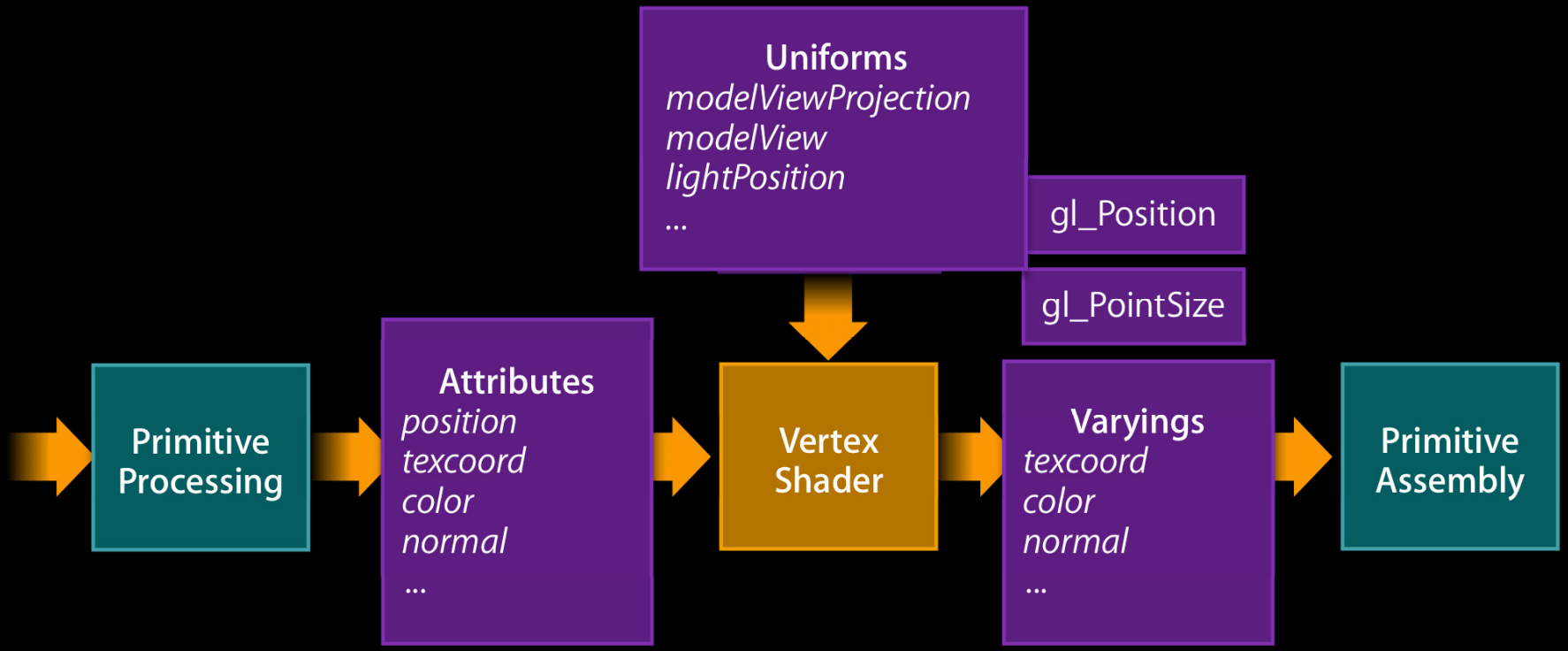
OpenGL ES 2.0

Vertex Shader

- Position and normal transformation
- Texture coordinate transformation
- Lighting equation
- Skinning

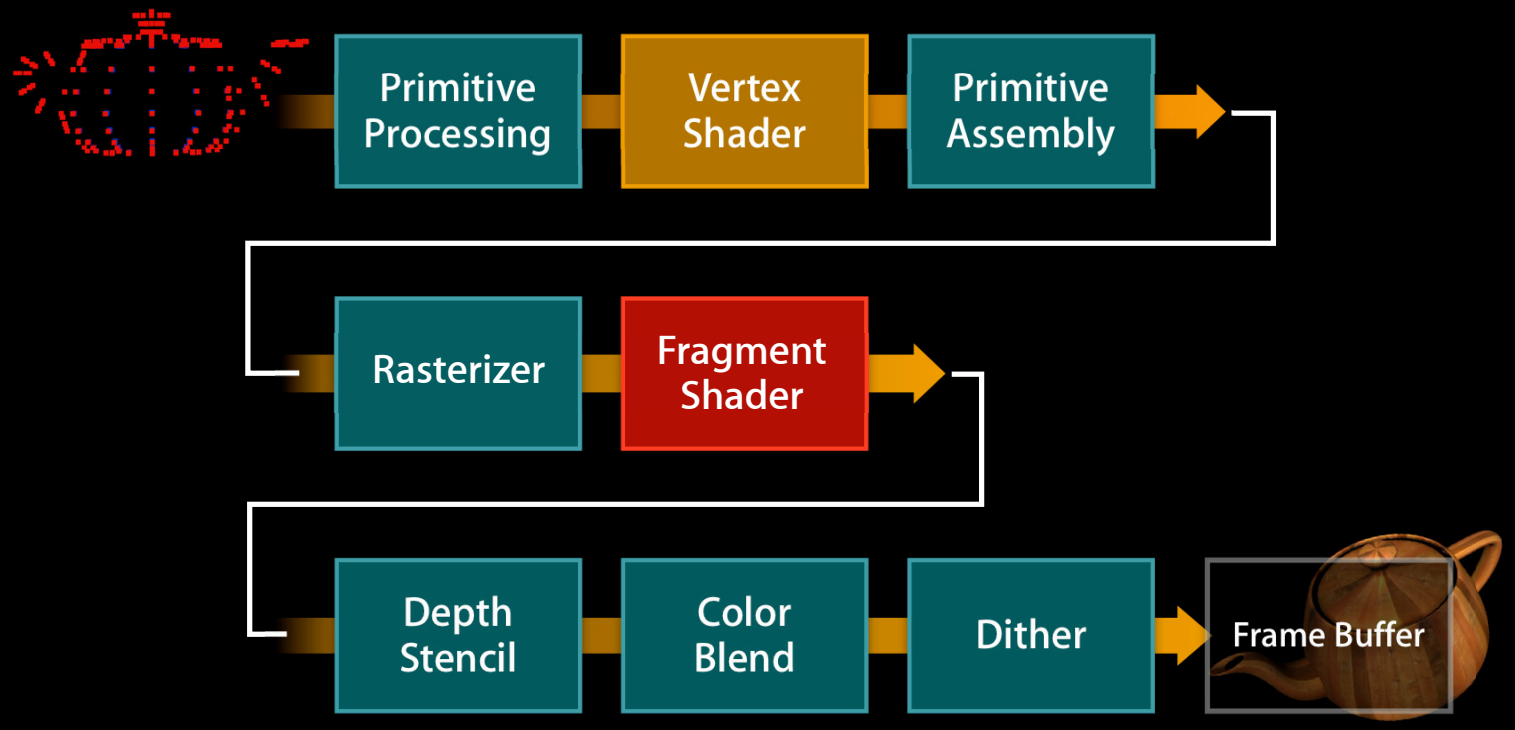
OpenGL ES 2.0

Vertex Shader I/O



OpenGL ES 2.0

Programmable graphics pipeline



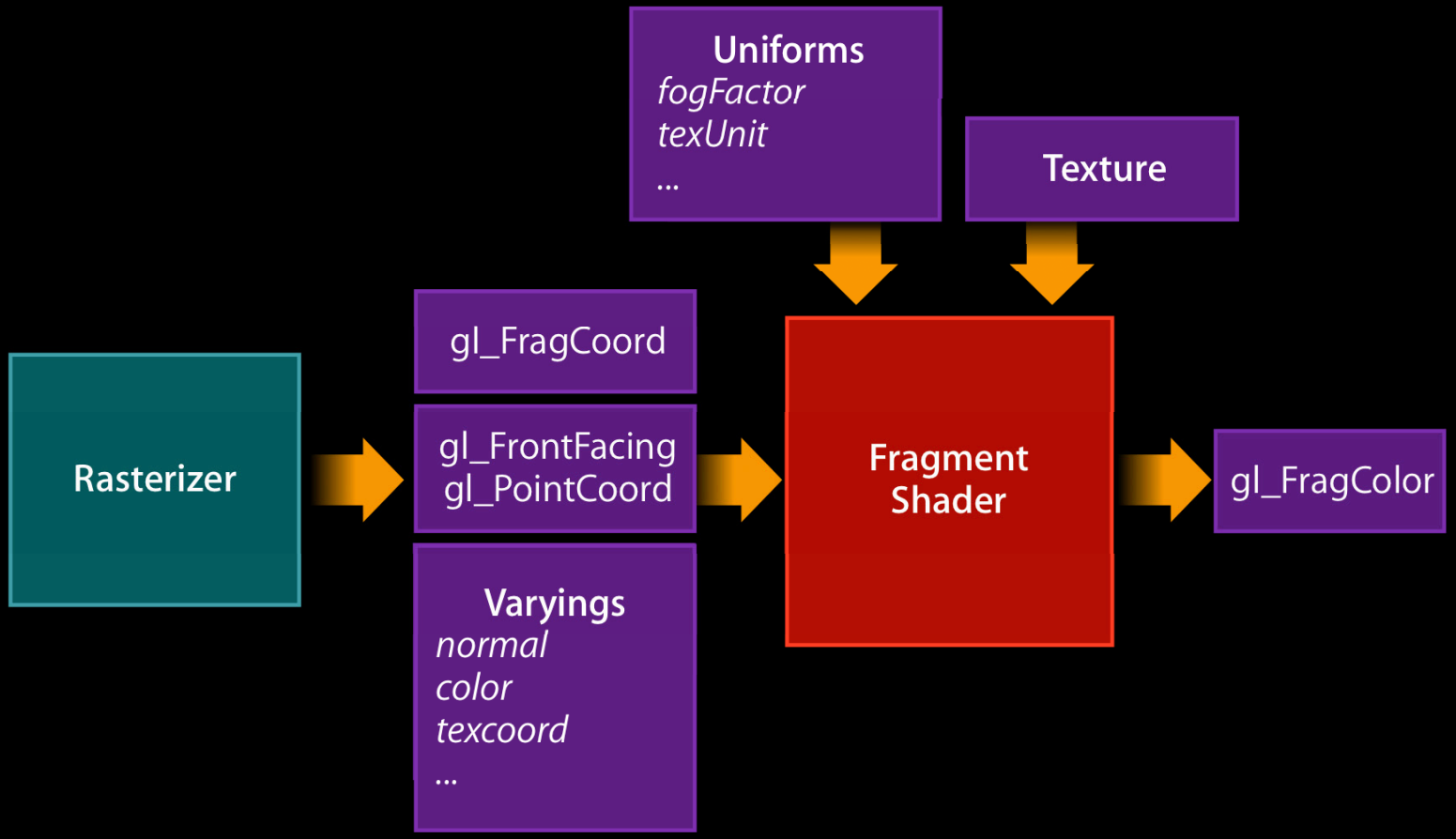
OpenGL ES 2.0

Fragment Shader

- Texture loading
- Texture environment
- Fog
- Alpha test

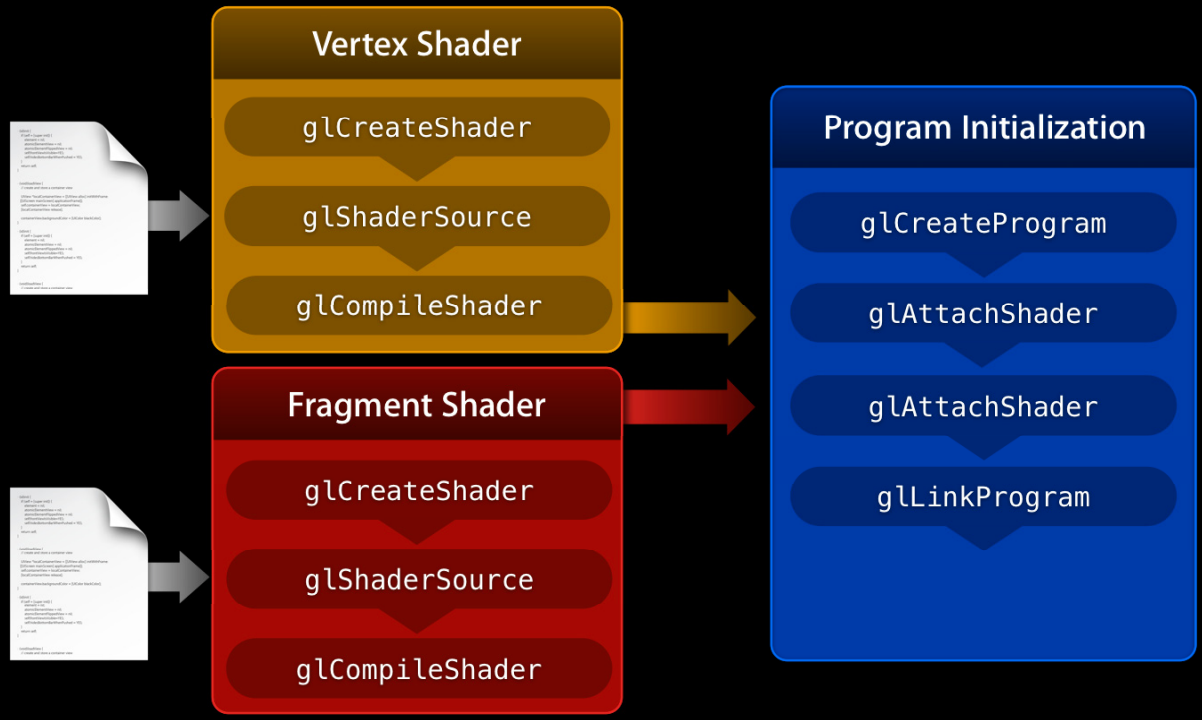
Fragment Shader I/O

Varyings, Uniforms, FragColor



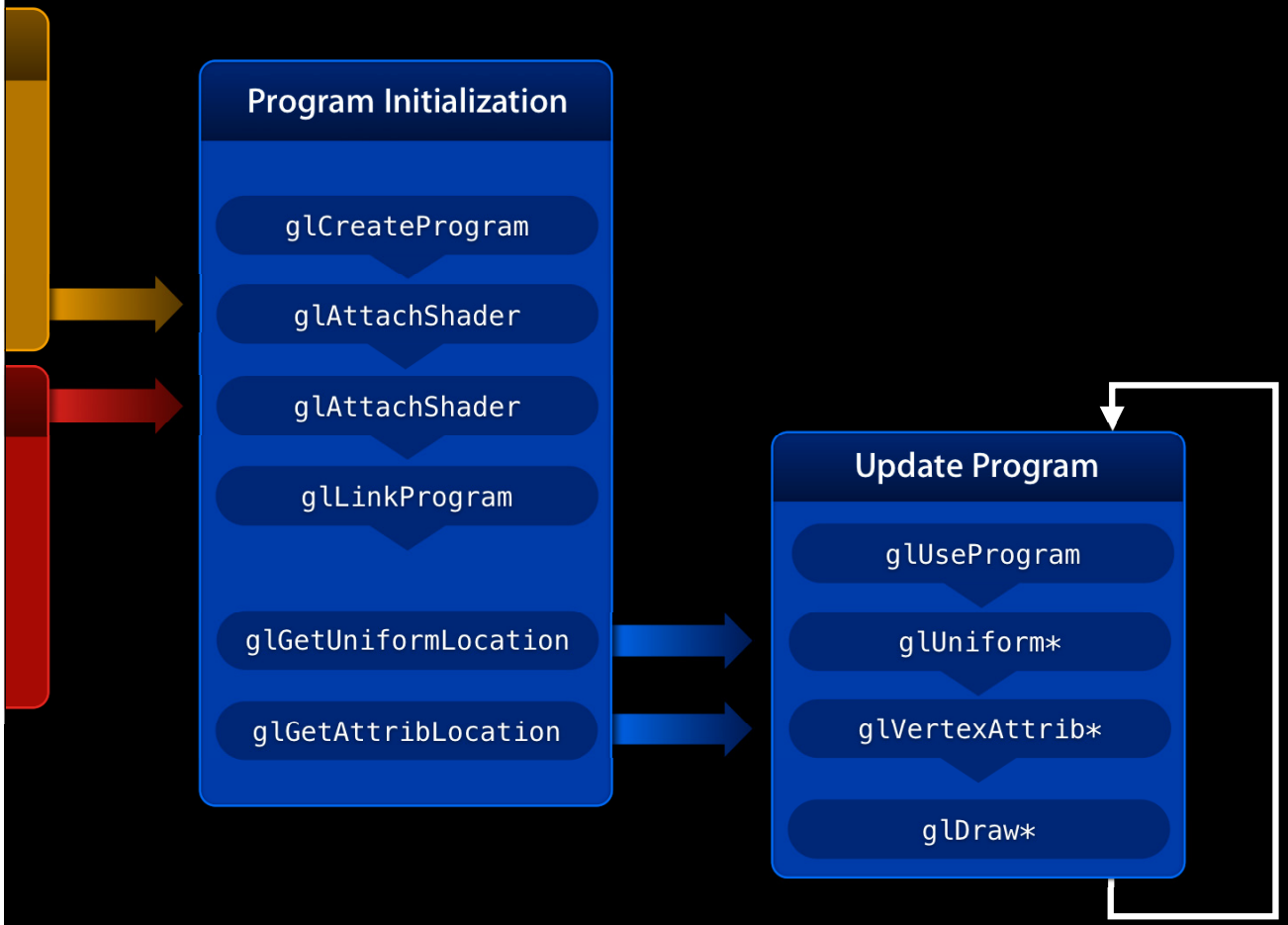
Shader Initial Setup

Compile, attach, link, use



Shader Initial Setup

Compile, attach, link, use



Rendering Static Environment

Texture mapping

Texture Atlas

Vertex Shader for Texture Mapping

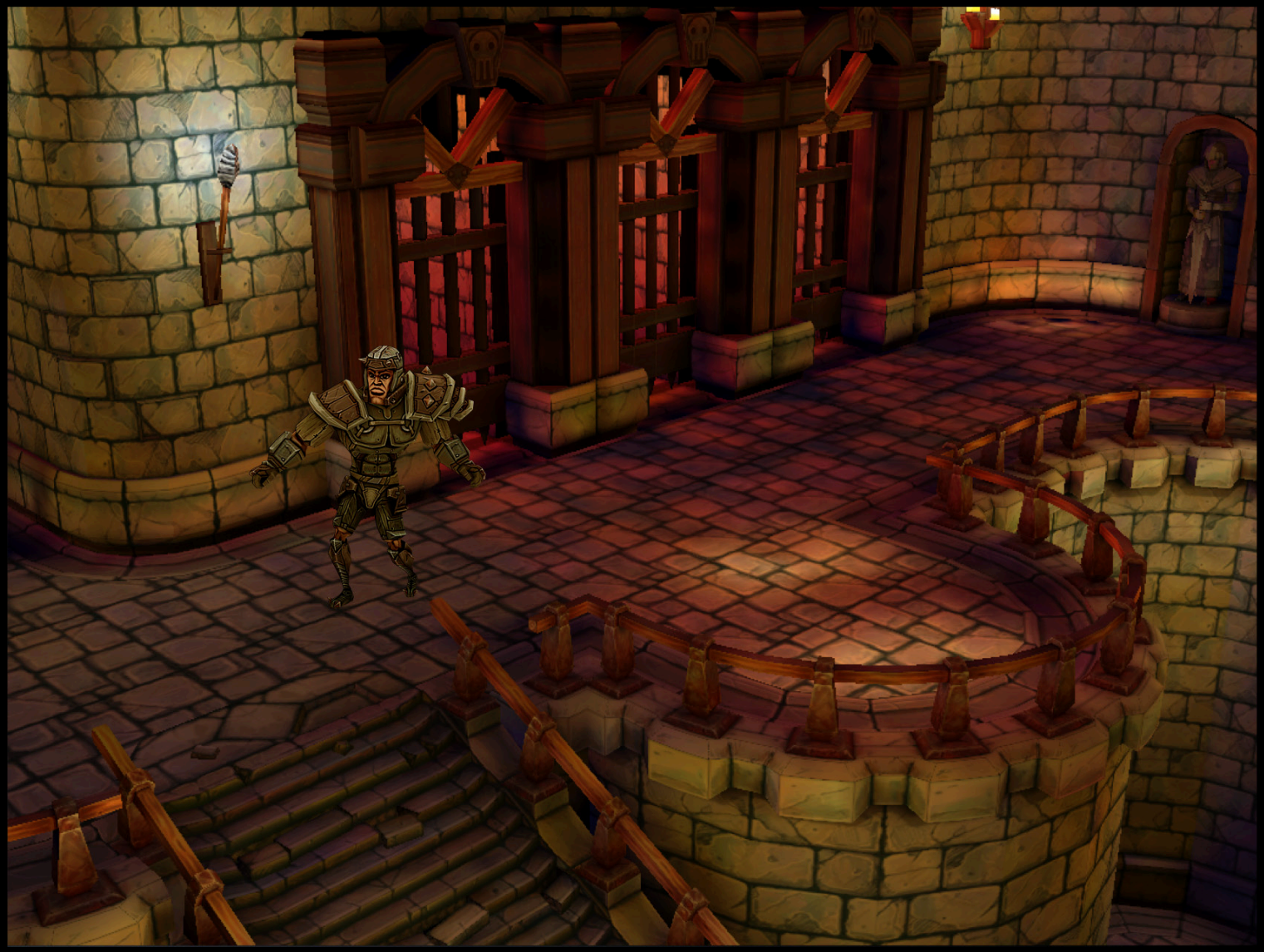
Position transformation

```
attribute vec4 a_position;
attribute vec2 a_texCoord;

uniform mat4 u_modelViewProjectionMatrix;

varying vec2 v_texCoord;

void main(void)
{
  gl_Position = u_modelViewProjectionMatrix * a_position;
  v_texCoord = a_texCoord;
}
```

Fragment Shader for Texture Mapping

Texture load

```
uniform sampler2D u_texture;

varying highp vec2 v_texCoord;

void main(void)
{
  gl_FragColor = texture2D(u_texture, v_texCoord);
}
```



Agenda

OpenGL ES 2.0

- Recap of graphics pipeline
- Basics of programmable shading

Real-time rendering techniques

- Skinning
- Lighting
- Shadowing

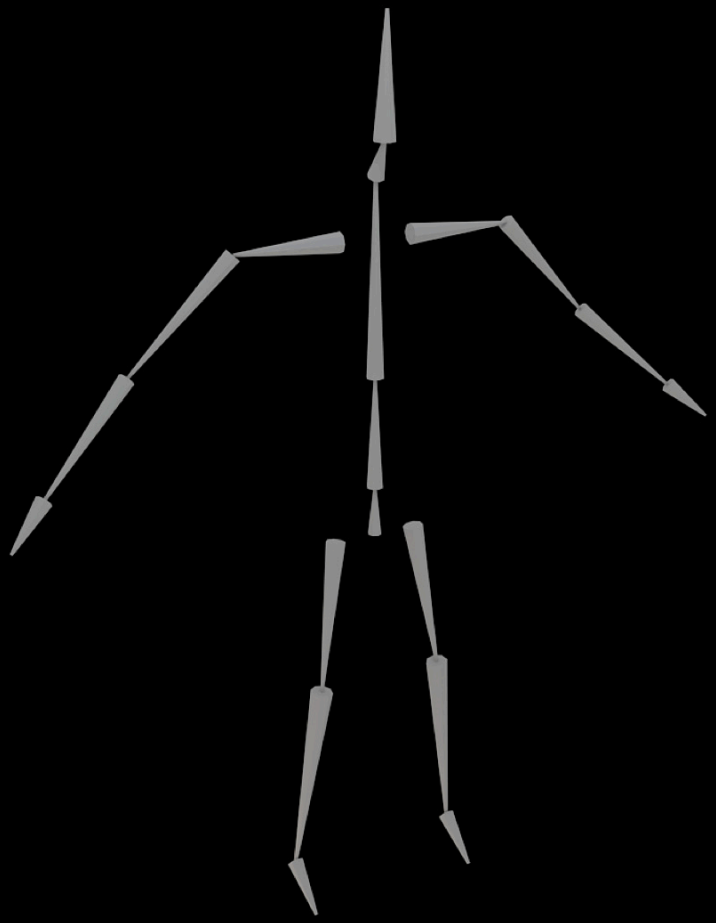
Agenda

- Skinning
- Lighting
- Shadowing

Real-Time Character Rendering

Skinning

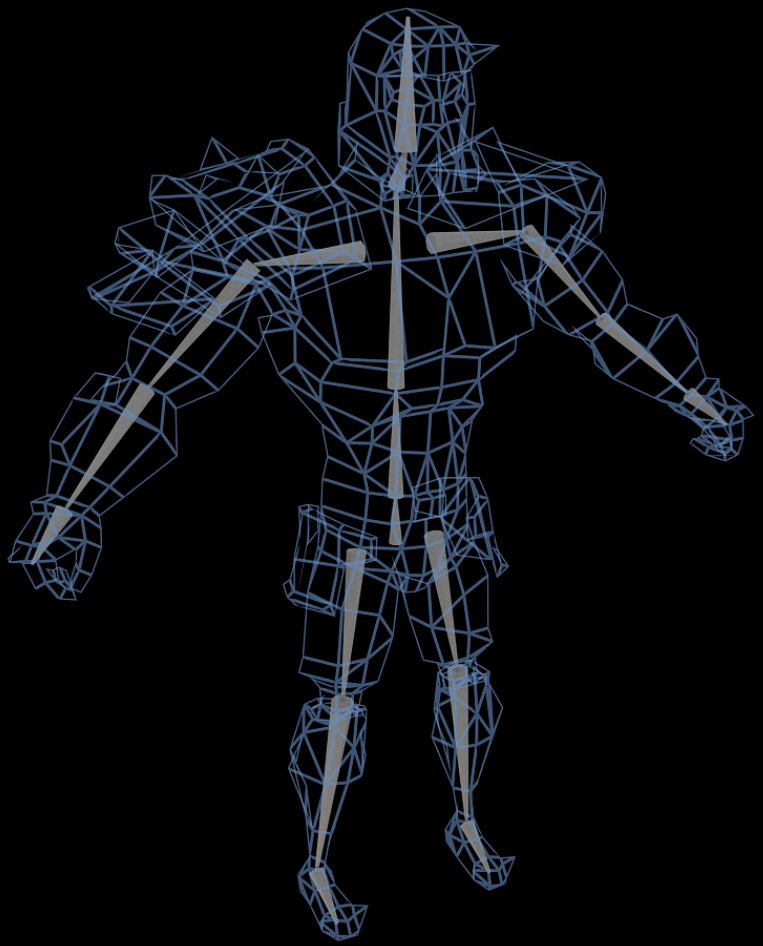
- Model skin deformation
- Based on movement of joints in articulated skeleton
- Smooth skinning (a.k.a. linear blend skinning)



Rendering Characters

Skeleton

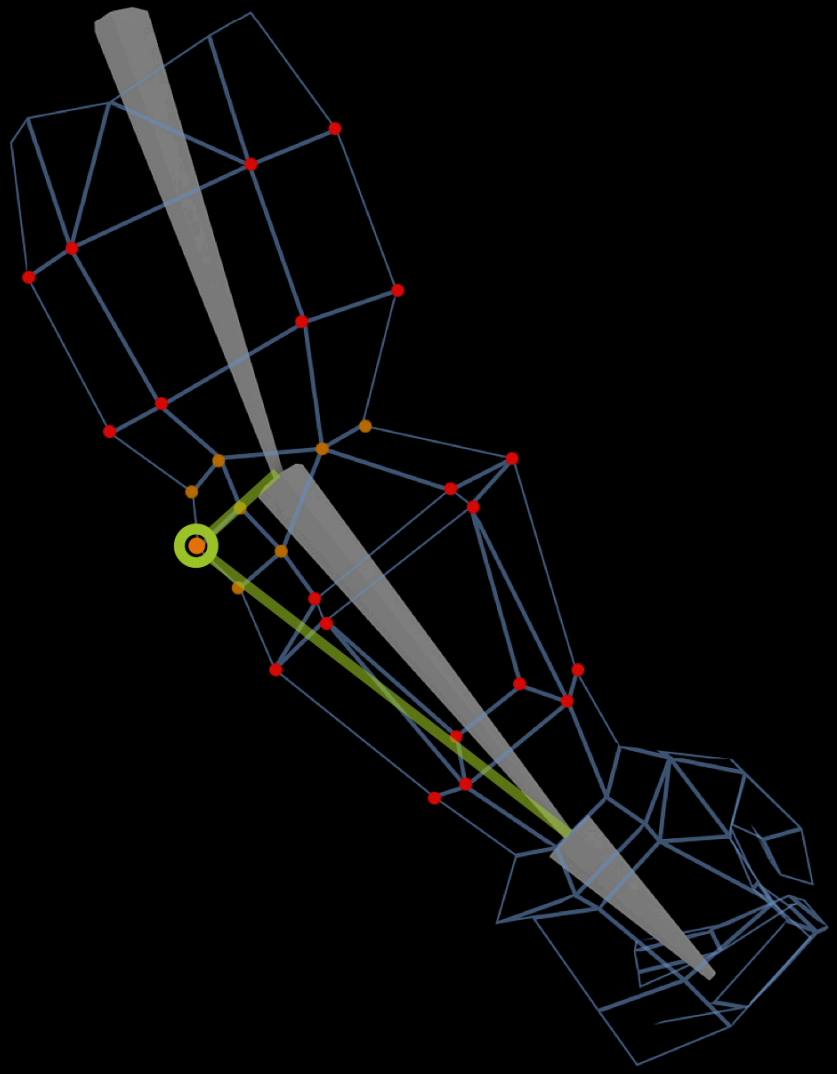
- Hierarchy of bones and joints
- Lets you animate your character



Rendering Characters

Linear smooth skinning

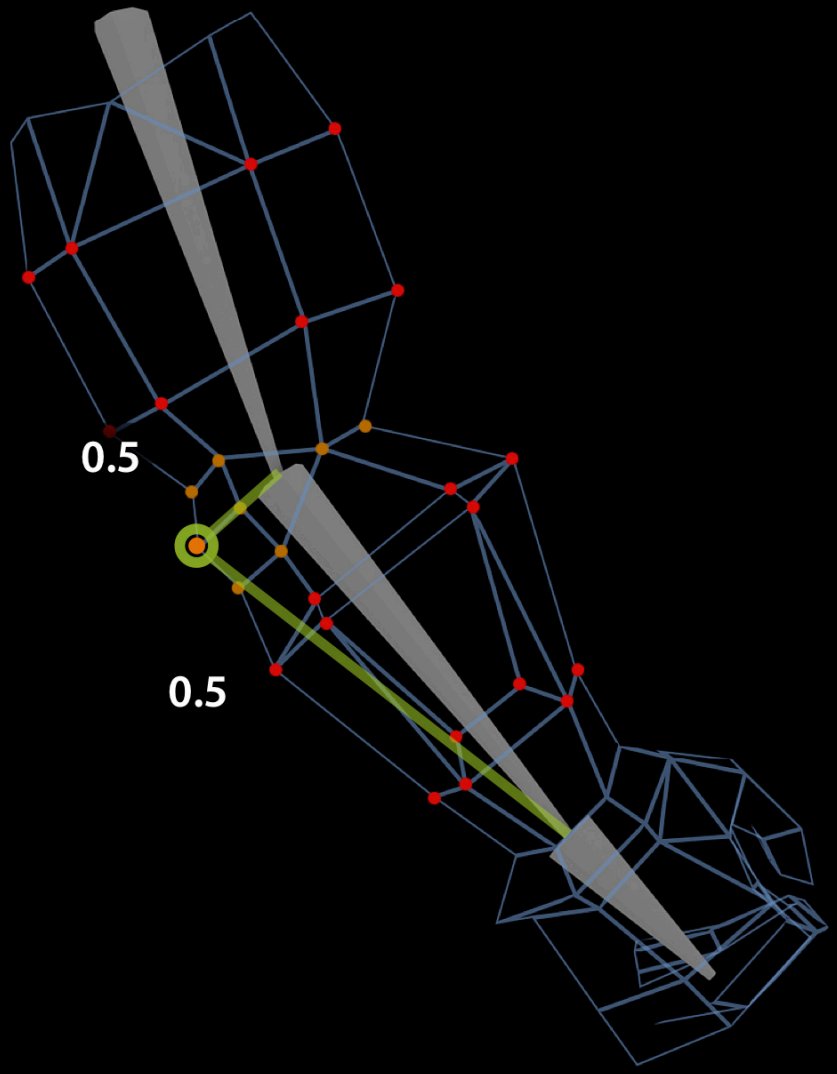
- Bind model to animated skeleton
- Each vertex is bound to N bones



Rendering Characters

Linear smooth skinning

- Bind model to animated skeleton
- Each vertex is bound to N bones



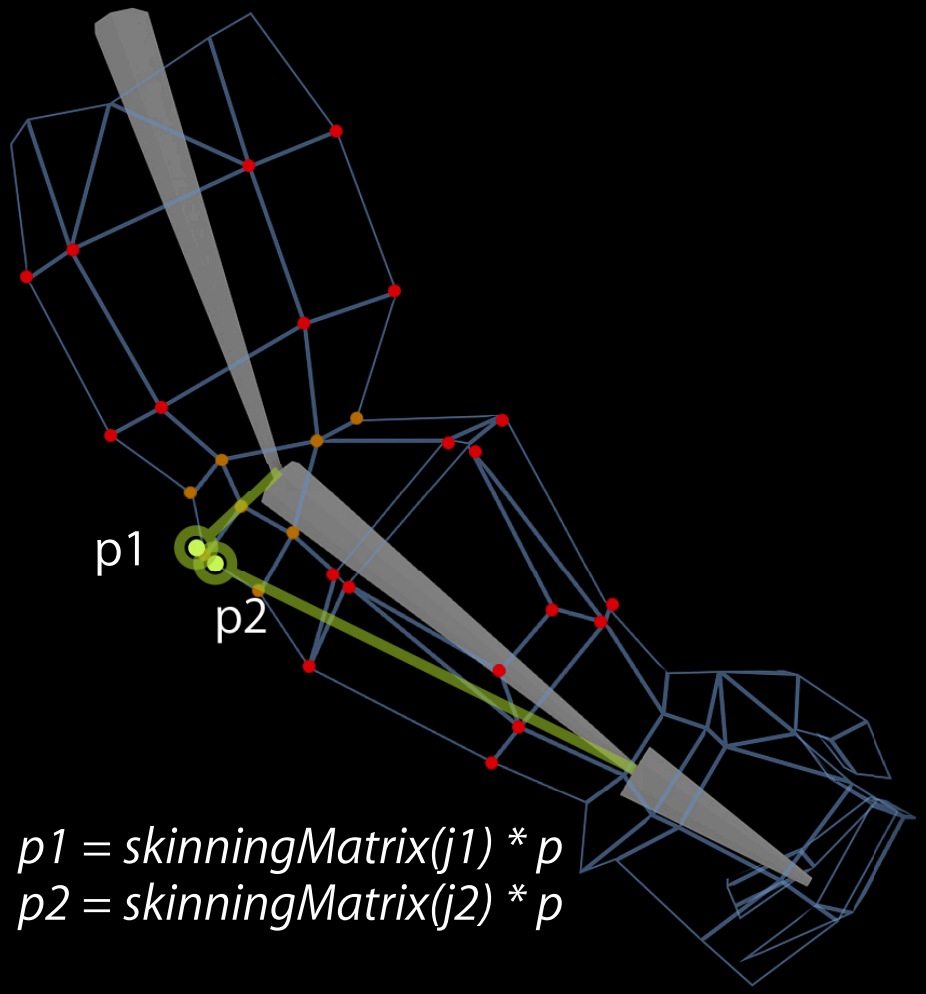
Rendering Characters

Linear smooth skinning

- Bind model to animated skeleton
- Each vertex is bound to N bones

Vertex attributes

- *weight(n)*
  - Represent influence of each bone
  - Closest bone has greatest influence



Rendering Characters

Linear smooth skinning

- Bind model to animated skeleton
- Each vertex is bound to N bones

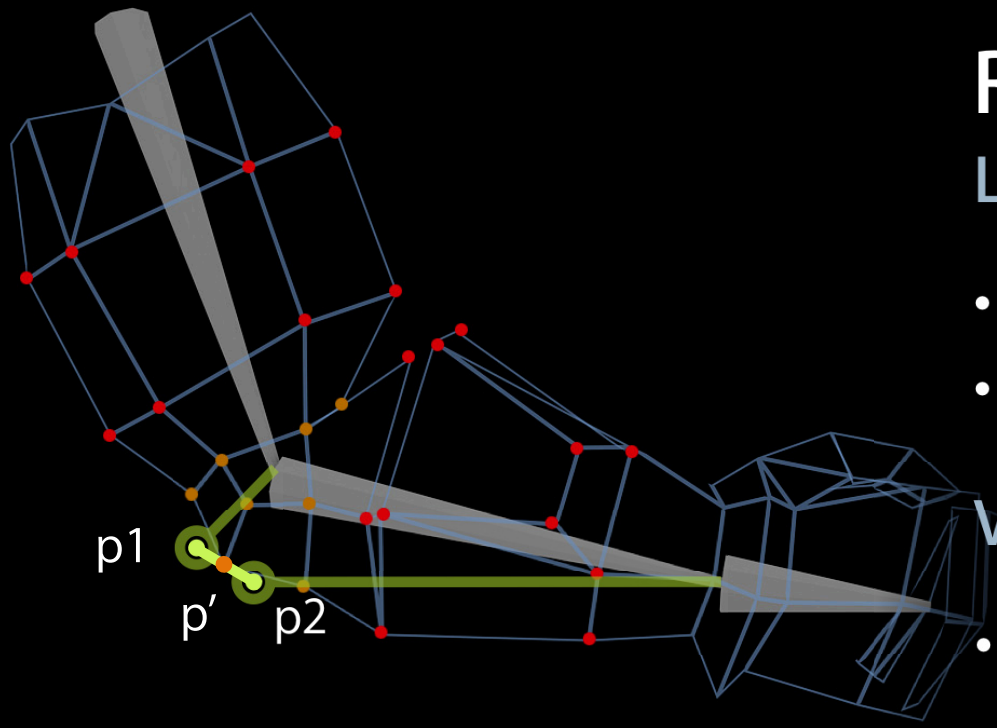
Vertex attributes

- *weight(n)*
  - Represent influence of each bone
  - Closest bone has greatest influence
- *skinningMatrix(n)*: combines
  - Transformation matrix for joint n
  - Position of joint relative to mesh

Rendering Characters

Linear smooth skinning

- Bind model to animated skeleton
- Each vertex is bound to N bones



*p1 = skinningMatrix(j1) * p*
*p2 = skinningMatrix(j2) * p*

*p' = weight(j1) * p1 + weight(j2) * p2*

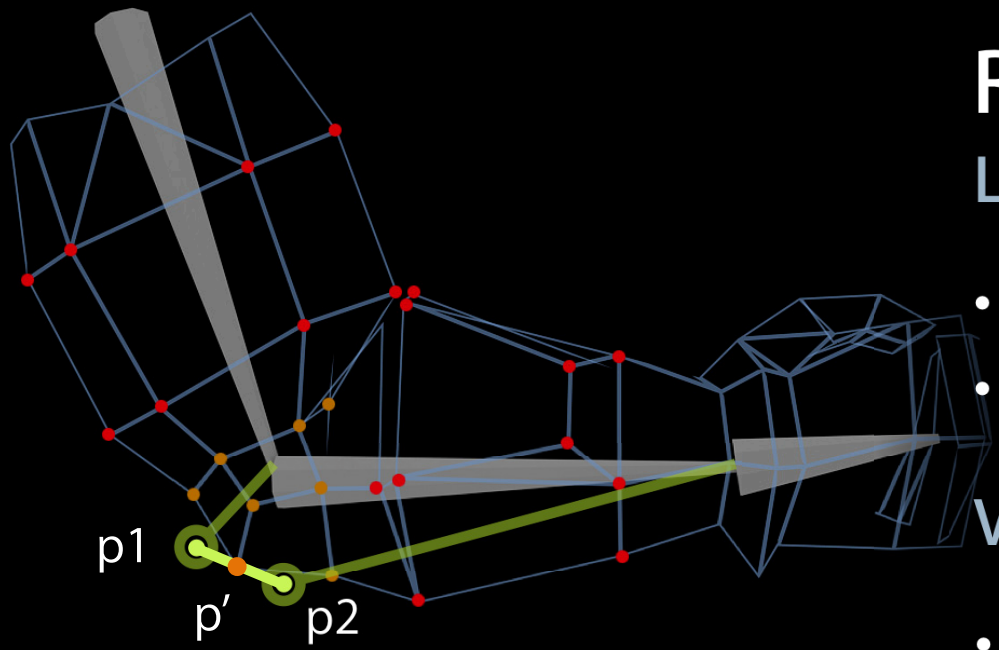
Vertex attributes

- *weight(n)*
  - Represent influence of each bone
  - Closest bone has greatest influence
- *skinningMatrix(n)*: combines
  - Transformation matrix for joint n
  - Position of joint relative to mesh

Rendering Characters

Linear smooth skinning

- Bind model to animated skeleton
- Each vertex is bound to N bones

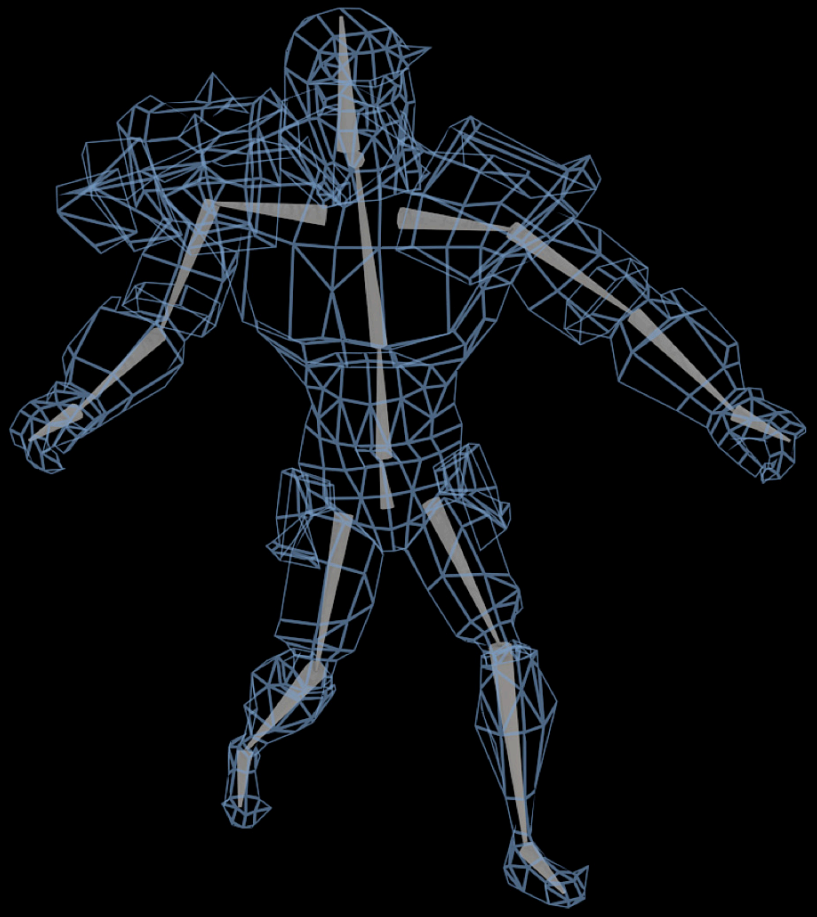


*p1 = skinningMatrix(j1) * p*
*p2 = skinningMatrix(j2) * p*

*p' = weight(j1) * p1 + weight(j2) * p2*

Vertex attributes

- *weight(n)*
  - Represent influence of each bone
  - Closest bone has greatest influence
- *skinningMatrix(n)*: combines
  - Transformation matrix for joint n
  - Position of joint relative to mesh



Rendering Characters

Linear smooth skinning

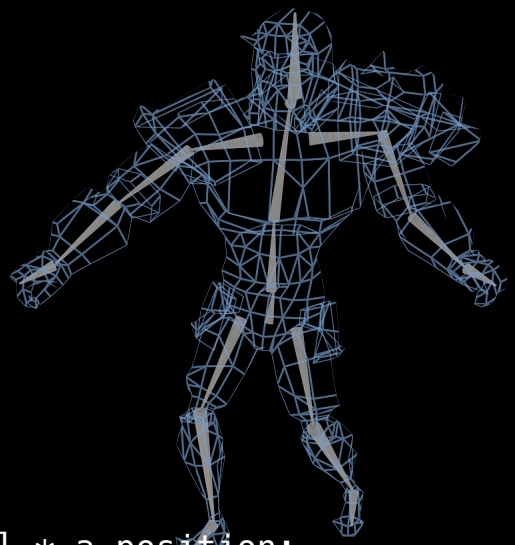
- Bind model to animated skeleton
- Each vertex is bound to N bones

Vertex attributes

- *weight(n)*
  - Represent influence of each bone
  - Closest bone has greatest influence
- *skinningMatrix(n)*: combines
  - Transformation matrix for joint n
  - Position of joint relative to mesh

Textbook Vertex Shader for Skinning

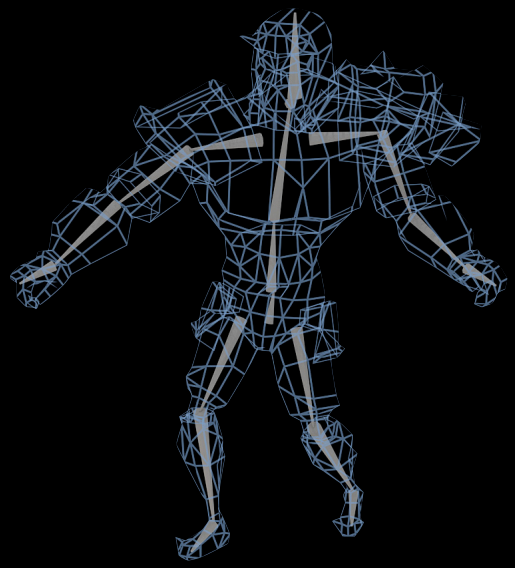
```
attribute vec4 a_position;
attribute float a_joint[N];
attribute float a_weight[N];

uniform mat4 u_skinningMatrix[JOINT_COUNT];
uniform mat4 u_modelViewProjectionMatrix;

void main(void)
{ ...
  p = vec4(0.0);
  for(i=0; i<N; i=i+1)
  {
    if(a_joint[i] != -1.0)
    {
      p = p + a_weight[i] * u_skinningMatrix[int(a_joint[i])] * a_position;
    }
  }
  gl_Position = u_modelViewProjectionMatrix * p;
  ...
}
```



Optimized Vertex Shader for Skinning

Assumption: Vertex always attached to two bones

```
attribute vec4 a_position;
attribute float a_joint1, a_joint2;
attribute float a_weight1, a_weight2;

uniform mat4 u_skinningMatrix[JOINT_COUNT];
uniform mat4 u_modelViewProjectionMatrix;



void main(void)
{
    vec4 p1 = u_skinningMatrix[int(a_joint1)] * a_position;
    vec4 p2 = u_skinningMatrix[int(a_joint2)] * a_position;
    vec4 p = p1 * a_weight1 + p2 * a_weight2;

    gl_Position = u_modelViewProjectionMatrix * p;

}
```



Optimized Vertex Shader for Skinning

Assumption: Vertex always attached to two bones

```
attribute vec4 a_position;
attribute float a_joint1, a_joint2;
attribute float a_weight1, a_weight2;

uniform mat4 u_skinningMatrix[JOINT_COUNT];
uniform mat4 u_modelViewProjectionMatrix;

attribute vec2 a_textureCoord;
varying vec2 v_textureCoord;

void main(void)
{
    vec4 p1 = u_skinningMatrix[int(a_joint1)] * a_position;
    vec4 p2 = u_skinningMatrix[int(a_joint2)] * a_position;
    vec4 p = p1 * a_weight1 + p2 * a_weight2;

    gl_Position = u_modelViewProjectionMatrix * p;

    v_textureCoord = a_textureCoord;
}
```

Demo

Agenda

- Skinning
- Lighting
- Shadowing

Real-Time Lighting

Unlit world

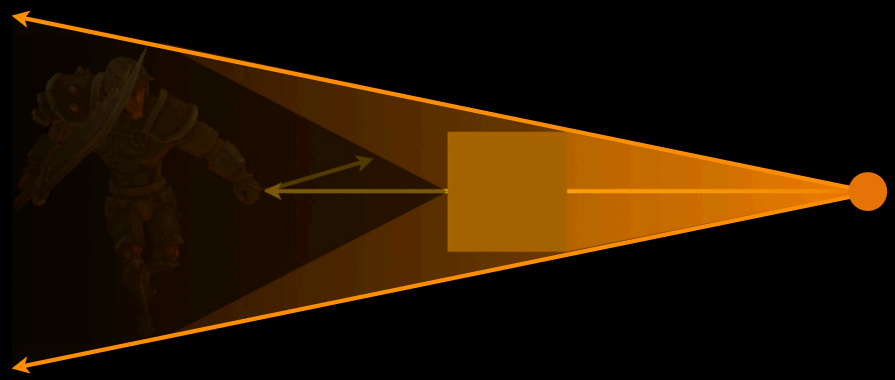
Real-Time Lighting

Fully lit and shadowed world

Light Contribution

Determined by three factors

- Distance
- Direction
- Occlusion



Lighting Content

Static vs. dynamic

OpenGL Light Model

Traditional model

- Accounts for
  - Distance
    - Attenuation modes and factors
  - Direction
    - Light vector and geometry normals
- Supports all static and dynamic geometries and lights

OpenGL Light Model

Diffuse light contribution

Linear Attenuation, Per-Vertex Lighting

Vertex Shader

```
attribute vec3 a_normal;
attribute vec4 a_position;
```

```
uniform vec4  u_lightColor;
uniform vec3  u_lightPosition;
uniform float u_linearAttenuation;
uniform mat4  u_modelViewProjectionMatrix;
```

```
varying vec4 v_lightColor;
```

```
void main(void)
{
  // Transform position
  // Linear attenuation
  // Direction factor
  // Light contribution accounting for direction and distance
}
```

Linear Attenuation, Per-Vertex Lighting

Vertex Shader

```
void main(void)
{
  // Transform position
  gl_Position = u_modelViewProjectionMatrix * a_position;

  // Linear attenuation
  vec3 lightVector = u_lightPosition - a_position.xyz;
  float distance = length(lightVector);
  float attenuation = 1.0 / (u_linearAttenuation * distance);

  // Direction factor
  lightVector = normalize(lightVector);
  float directionFactor = max(0.0, dot(a_normal, lightVector));

  // Light contribution accounting for direction and distance
  v_lightColor = u_lightColor * directionFactor * attenuation;
}
```

OpenGL Light Model

Considerations

- Computationally expensive
- Computed for each frame
- Visual improvement
  - Accounts for direction and distance
  - Does not account for occlusion

Prebaking Static Light Contribution

Static lights and static geometry

- Light contribution is known after world construction
  - Computed during world export
  - Create world space or per-object lightmaps
  - Atlas lightmaps together
- Accounts for
  - Distance
  - Direction
  - Occlusion

Per-Object Lightmaps

Considerations for prebaking

- Works only for static geometry
- Uses radiosity, direct illumination, or similar algorithm
- Unique texture for geometry based on world X, Y, Z

Applying Per-Object Lightmaps

Static lights and static geometry

Applying Per-Object Lightmaps

Static lights and static geometry

Applying Per-Object Lightmaps

Fragment Shader

```
uniform lowp sampler2D u_diffuseTexture;
uniform lowp sampler2D u_lightmapTexture;

varying highp vec2 v_diffuseUVs;
varying highp vec2 v_lightmapUVs;

void main(void)
{
  lowp vec4 lightColor   = texture2D(u_lightmapTexture, v_lightmapUVs);
  lowp vec4 diffuseColor = texture2D(u_diffuseTexture,  v_diffuseUVs);
  gl_FragColor = diffuseColor * lightColor;
}
```

Prebaked Lighting

Static lights and dynamic geometry

- Approximate contribution for dynamic geometry
  - Per-object lightmaps do not work
- Compute contribution for a world X, Y, Z
  - Creates a lightmap in world space
- Accounts for
  - Distance
  - Occlusion

World Space Lightmaps

Approximating in 2.5D

- Apply single 2D top-down lightmap
- Transform X, Y, Z (world) to U,V (lightmap) coordinates
  - Flatten X, Y, Z
  - Scale and translate
- Transform model X, Y, Z to U,V in vertex shader

Applied World Space Lightmaps

Static lights and dynamic geometry

World Space Lightmaps

Vertex Shader: Skinning with 2D lightmaps

```
...
uniform mat4 u_lightmapProjectionMatrix;
varying vec2 v_lightmapUVs;
void main(void)
{
  vec4 p1 = u_skinningMatrix[int(a_joint1)] * a_position;
  vec4 p2 = u_skinningMatrix[int(a_joint2)] * a_position;
  vec4 p = p0 * a_weight1 + p1 * a_weight2;

  gl_Position = u_modelViewProjectionMatrix * p;

  v_diffuseUVs = a_diffuseUVs;

  v_lightmapUVs = vec2(u_lightmapProjectionMatrix * p);
}
```

World Space Lightmaps

Approximating in 2.5D

- Single 2D lightmap
- Fast and efficient
- Straightforward setup
- No direction contribution

World Space Lightmaps

Artifacts due to missing direction contribution

Combining Direction with Lightmaps

A hybrid approach

- OpenGL lighting model determines direction factor
- Modulate light contribution by direction factor
- Requires a lightmap per light

World Space Lightmaps

Visualizing per-light lightmaps

World Space Lightmaps with Direction

Per-light contribution for dynamic geometry

World Space Lightmaps with Direction

Putting it all together

World Space Lightmaps with Direction

Vertex Shader: Skinned lightmaps with direction

```
void main(void)
{

  ...

}
```

```
void main(void)
{
  vec4 p0 = u_skinningMatrix[int(a_joint1)] * a_position;
  vec4 p1 = u_skinningMatrix[int(a_joint2)] * a_position;
  vec4 p = p0 * a_weight1 + p1 * a_weight2;

  vec4 n0 = u_skinningMatrix[int(a_joint1)] * a_normal;
  vec4 n1 = u_skinningMatrix[int(a_joint2)] * a_normal;
  vec3 n = n0.xyz * a_weight1 + n1.xyz * a_weight2;
  n = normalize(u_modelViewMatrix * n);

  vec3 lightVector = normalize(u_lightPosition[0] - p.xyz);
  v_lightFactor.x = max(0.0, dot(n, lightVector));

  lightVector = normalize(u_lightPosition[1] - p.xyz);
  v_lightFactor.y = max(0.0, dot(n, lightVector));

  lightVector = normalize(u_lightPosition[2] - p.xyz);
  v_lightFactor.z = max(0.0, dot(n, lightVector));

  v_lightmapUVs = vec2(u_lightmapProjection * p);

  v_diffuseUVs = a_diffuseUVs;

  gl_Position = u_modelViewProjectionMatrix * p;
}
```

World Space Lightmaps with Direction

Fragment Shader: Mixing direction and lightmaps

```
uniform lowp sampler2D u_diffuseTexture;
uniform lowp sampler2D u_lightmapTexture[3];
varying vec2 v_lightmapUVs;
varying vec2 v_diffuseUVs;
varying vec3 v_lightFactor;

void main(void)
{
  lowp vec4 lightColor, diffuseColor;

  lightColor   = texture2D(u_lightmapTexture[0], v_lightmapUVs) * v_lightFactor.x;
  lightColor  += texture2D(u_lightmapTexture[1], v_lightmapUVs) * v_lightFactor.y;
  lightColor  += texture2D(u_lightmapTexture[2], v_lightmapUVs) * v_lightFactor.z;

  diffuseColor = texture2D(u_diffuseTexture, v_diffuseUVs);

  gl_FragColor = diffuseColor * lightColor;
}
```

World Space Lightmaps with Direction

Considerations

- Nuanced visual improvement
  - Avoids artifacts
- Increased GPU and memory cost
- Fully accounts for
  - Distance
  - Direction
  - Occlusion

Demo

Lighting

Summary

- Traditional OpenGL light model
  - Works for all content
- Static lights and static geometry
  - Use per-object lightmaps
- Static lights and dynamic geometry
  - Single world space lightmap
  - Multiple world space lightmaps with direction

Agenda

- Skinning
- Lighting
- Shadowing

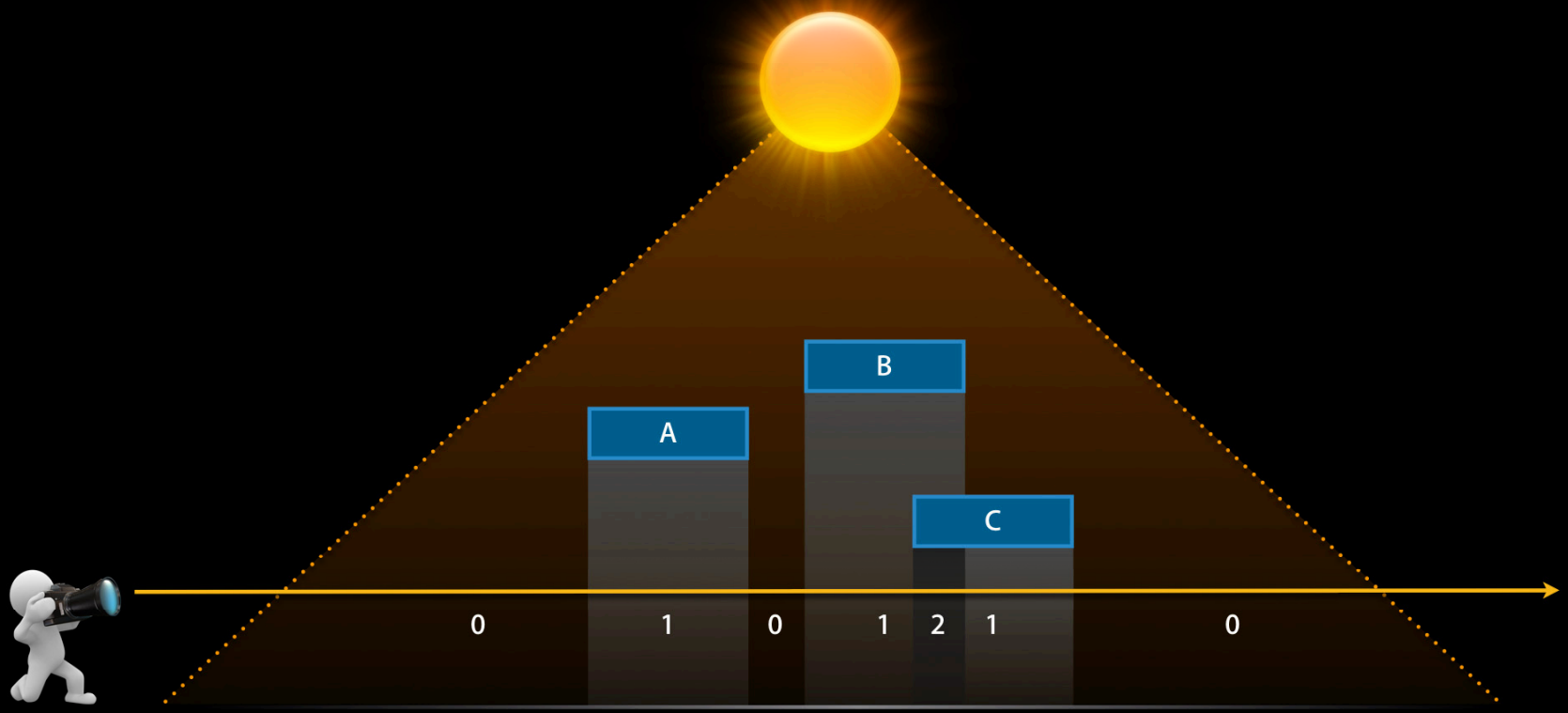
Shadowing

Shadow volumes

- Works well with per-light lightmaps
- Determines shadowed or lit per pixel
- Fully supports dynamic content
  - Shadows world
  - Shadows self

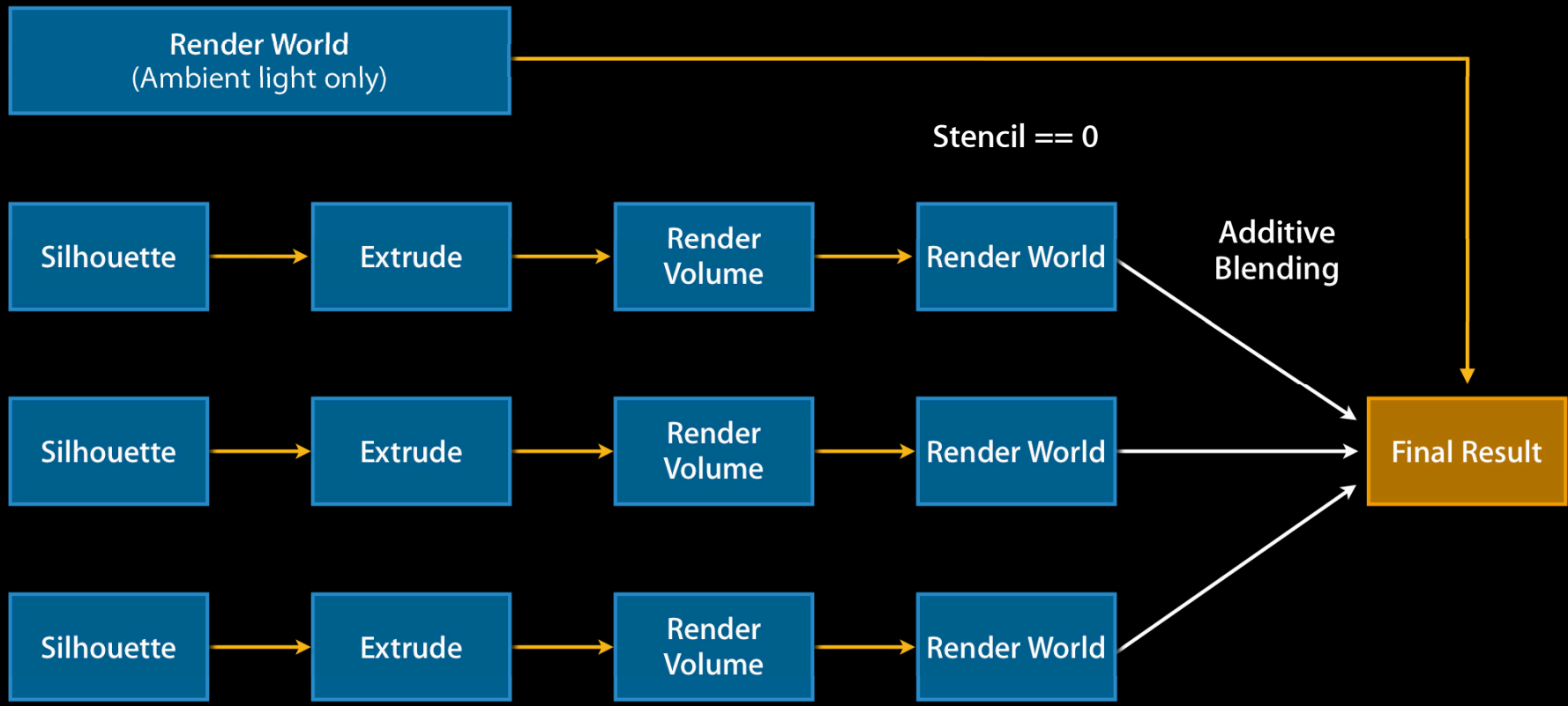
Counting Shadows

How it works



Using the Stencil Buffer

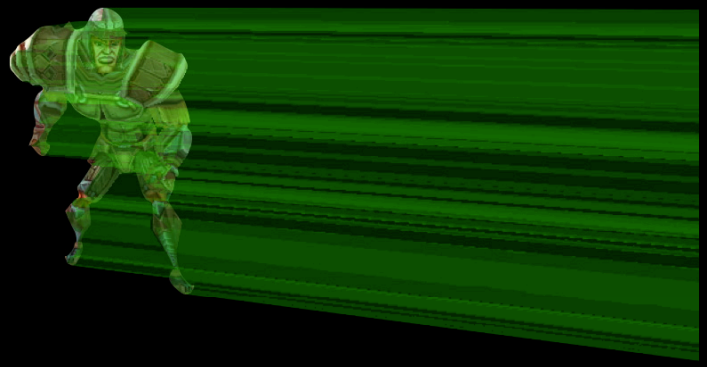
Overview



Using the Stencil Buffer

Generating shadow volumes

- Silhouette determination
  - Edges—opposite facing triangles
  - Facing—dot product of light vector and normal
- Extrude silhouette
  - Project geometry using `w = 0` for infinitely far vertices



Using the Stencil Buffer

Setting up the counting buffer

- Render shadow volumes from camera's view
  - Stencil increments when entering a volume
  - Stencil decrements when exiting a volume
- OpenGL ES 2.0 supports separate stencil
  - Set INCR_WRAP for GL_FRONT
  - Set DECR_WRAP for GL_BACK

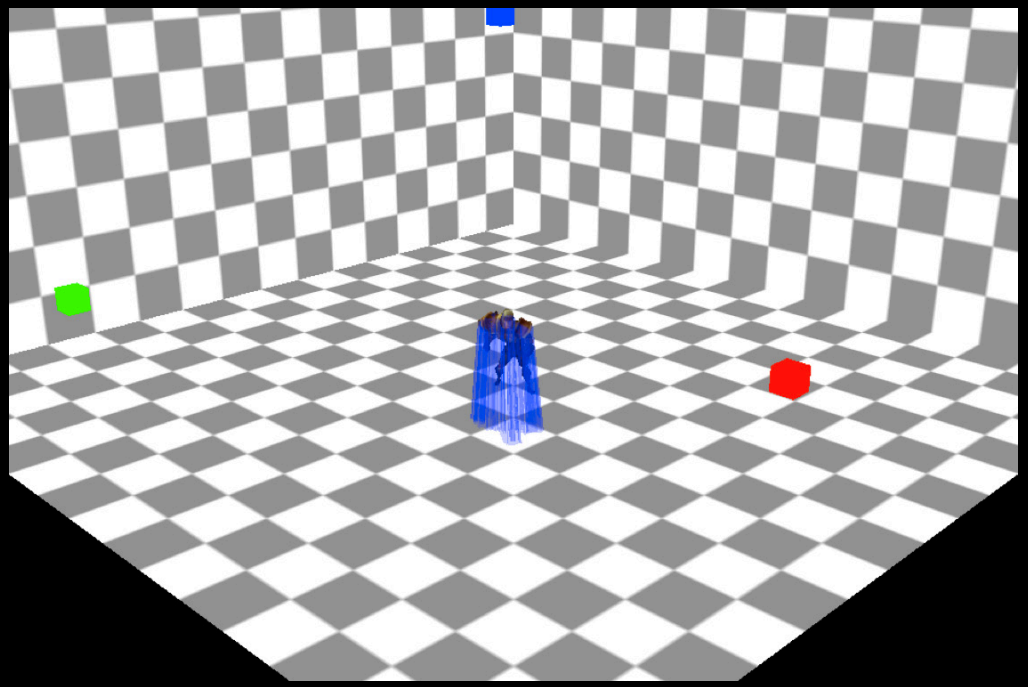
Using the Stencil Buffer

Lighting using the counting buffer

- If count is non-zero, the pixel is shadowed
- Lighting pixels
  - Set stencil test to EQUAL and stencil value to zero
  - Set depth test to EQUAL
  - Set blending to additive
  - Draw world with single light contribution

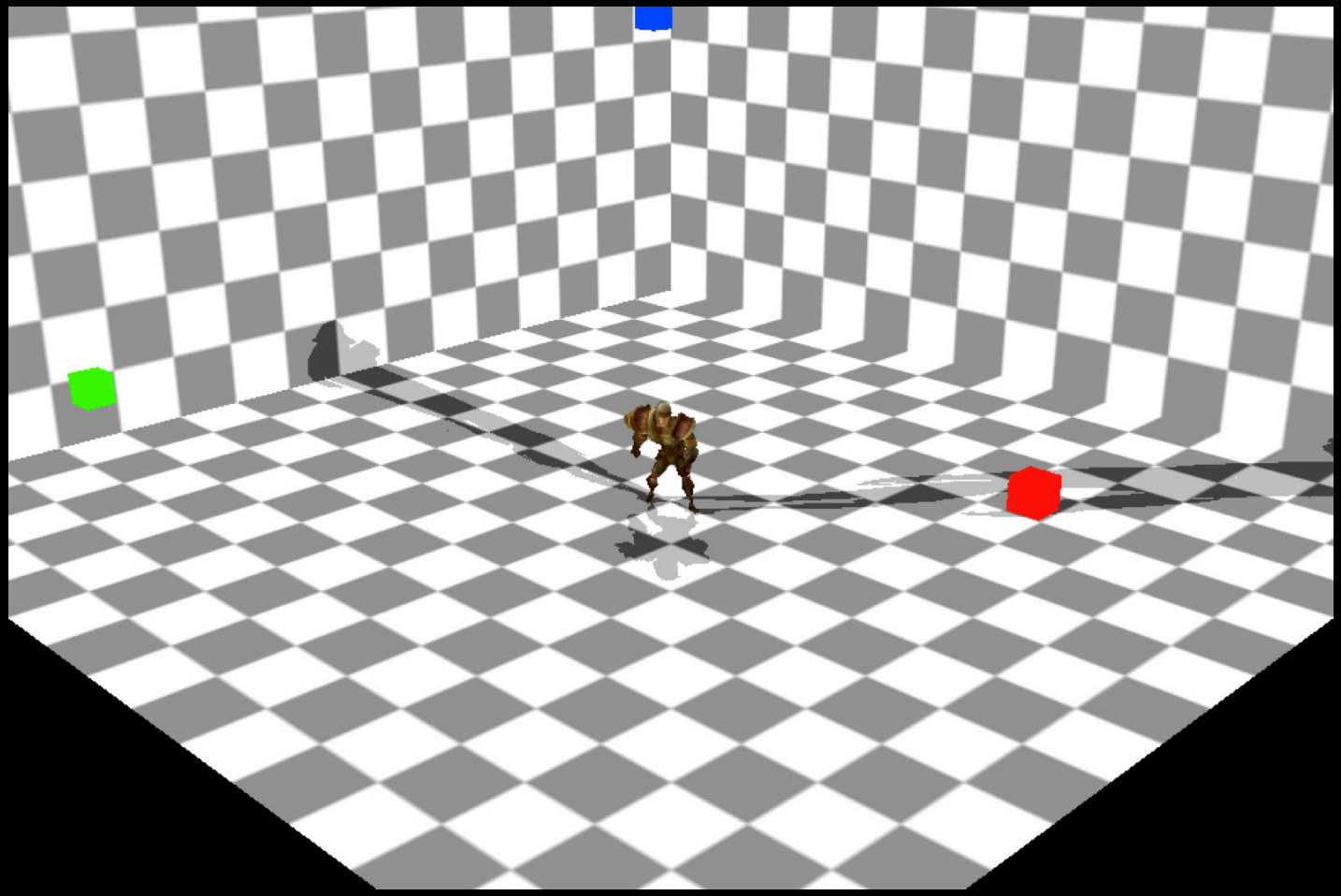
Shadow Volumes

Visualized per light



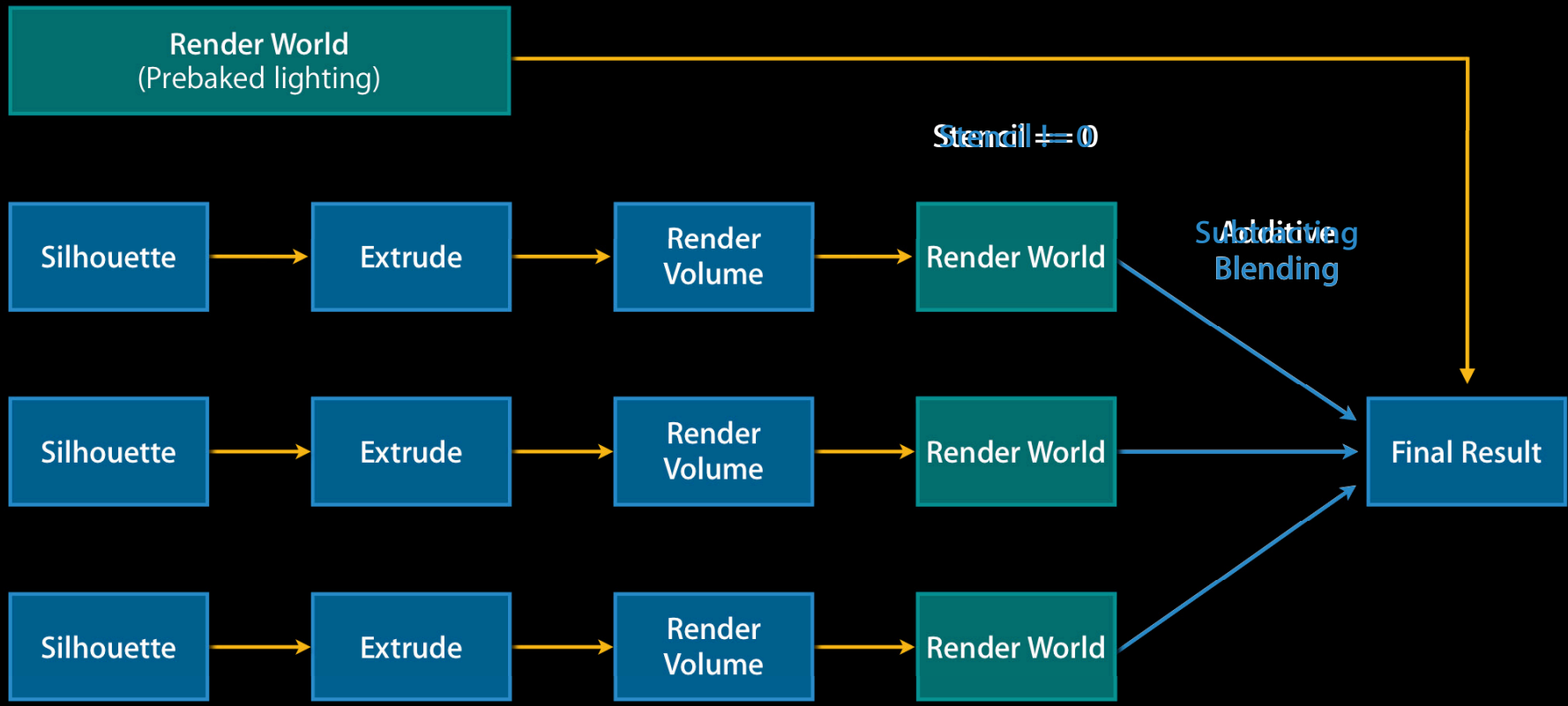
Shadow Casting

Visualized per light



Shadowing with Prebaked Content

How is it different?



Prebaked World with Shadow Volumes

Visualized per light

Shadowing with Prebaked Lights

- Visualized per light

Shadowing with Prebaked Lights

Shadowing with Prebaked Lights

Considerations

- Increased GPU cost
  - Shadow volumes incur vertex processing
  - Shadowed pixels incur fragment processing
- Prebaked lightmap contains all static shadows
  - Shadow volumes need only produce dynamic shadows
- Shadow volumes are well suited to PowerVR SGX
  - Efficient stencil and depth logic
  - Single pass

Demo

Agenda

OpenGL ES 2.0

- Recap of graphics pipeline
- Basics of programmable shading

Real-time rendering techniques

- Skinning
- Lighting
- Shadowing

Summary

Using OpenGL ES 2.0

- Precompute
  - Allows simplification of algorithms
  - Enables real-time rendering
- Choose appropriate algorithms
  - Optimize for performance
  - Improve visual quality

More Information

Allan Schaffer
Graphic and Game Technologies Evangelist
aschaffer@apple.com

Apple Developer Forums
http://devforums.apple.com

Related Sessions

OpenGL ES Overview for iPhone OS	Presidio Wednesday 2:00PM
OpenGL ES Tuning and Optimization	Presidio Wednesday 4:30PM
Game Design and Development for iPhone OS, Part 1	Presidio Tuesday 9:00AM
Game Design and Development for iPhone OS, Part 2	Presidio Tuesday 10:15AM
OpenGL for Mac OS X	Nob Hill Thursday 9:00AM
OpenGL Essential Design Practices	Pacific Heights Wednesday 11:30AM

Labs

OpenGL ES Lab	Graphics & Media Lab A Thursday 9:00AM–4:30PM
Game Design for iPhone OS Lab	Graphics & Media Lab A Friday 11:30AM–2:30PM


WWDC10