Texture Mapping Applications and Other Buffers

Prof. Vladlen Koltun
Computer Science Department
Stanford University

January 2011
Shadows

- Valuable cue of spatial relationships
- Increases realism
Shadows

- Valuable cue of spatial relationships
- Increases realism
Shadow mapping

- First pass: render the scene from the viewpoint of the light, store depth buffer as texture (shadow map)
- Second pass: project vertices into shadow map and compare depth values
Shadow mapping

- First pass details: can disable all rendering features that do not affect depth map.

- Second pass details: For vertices, use the light’s model-view and projection transforms to obtain \((u,v)\) coordinates in the shadow map and the depth \(w\) of the vertex. For fragments, perform perspective-correct interpolation.

- Compare \(w\) with value \(w'\) stored in \((u,v)\) in the shadow map. If \(w \leq w'\), perform lighting calculations with this light. Otherwise, do not.
• Numerical imprecision leads to self-shadowing

• Solution: add a bias $\varepsilon$. Change comparison from $w \leq w'$ to $w \leq w' + \varepsilon$

• Can use glPolygonOffset
Setting the bias

- Numerical imprecision leads to self-shadowing
- Solution: add a bias $\varepsilon$. Change comparison from $w \leq w'$ to $w \leq w' + \varepsilon$
- Can use `glPolygonOffset`

Too little

Too much

Just right
Shadow map aliasing

- Insufficient shadow map resolution leads to blocky shadows
- No easy solution. Should not filter depth values: leads to errors at object boundaries
- Percentage-closer filtering: filter comparison results
Other issues

• Additional rendering pass for each shadow-casting light

• Setting the “field of view” of the light. Can use spotlights, or a cube map (six shadow maps) for a point light.
Reflection mapping

- Render the scene from a single point inside the reflective object. Store rendered images as textures.
- Map textures onto object. Determine texture coordinates by reflecting view ray about the normal.
Cube mapping

- Render the scene six times, through six faces of a cube, with 90-degree field-of-view for each image.
- Store images in six textures, which represent an omni-directional view of the environment

Greene, 1986
Cube mapping

- To compute texture coordinates, reflect the view vector \( \mathbf{v} \) about the normal \( \mathbf{n} \): 
  \[
  \mathbf{r} = 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n} - \mathbf{v}
  \]
- The highest (in absolute value) coordinate of \( \mathbf{r} \) identifies which of the six maps we need. The texture coordinates in this map are obtained by normalizing the other two coordinates of \( \mathbf{r} \).
Sphere mapping

- Cube maps require maintaining six texture in memory
- Sphere mapping uses a single viewpoint-specific environment map, updated every frame
- Map depicts a perfectly reflective sphere viewed orthographically

Greene, 1986
Reflection mapping limitations

- Self-reflections not supported. A concave object will not reflect parts of itself.

- Environment map only correct for the point from which it was rendered. Good approximation for distant reflected objects, but can lead to substantial artifacts in general.
Bump mapping

- Simulates roughness ("bumpiness") of a surface without adding geometry
- Uses a two-dimensional height field (bump map) to perturb the normal during per-fragment shading calculations
- Limitation: silhouette is unaffected
Bump mapping

• Simulates roughness ("bumpiness") of a surface without adding geometry

• Uses a two-dimensional height field (bump map) to perturb the normal during per-fragment shading calculations

• Limitation: silhouette is unaffected

Blinn, SIGGRAPH 1978
\[ B = (u, v) \]

\[ P' = P + B(u, v)N \]
Bump mapping derivation

These are not really partial derivatives, but we follow Blinn’s original notation

\[ P_u = \frac{\partial P'}{\partial u} = \frac{\partial}{\partial u} (P + B(u, v)N) \]
\[ = P_u + \frac{\partial B}{\partial u} N + B(u, v) \frac{\partial N}{\partial u} \]
\[ \approx P_u + \frac{\partial B}{\partial u} N \]

\[ P_v = \frac{\partial P'}{\partial v} = \frac{\partial}{\partial v} (P + B(u, v)N) \]
\[ = P_v + \frac{\partial B}{\partial v} N + B(u, v) \frac{\partial N}{\partial v} \]
\[ \approx P_v + \frac{\partial B}{\partial v} N \]

The normals are always appropriately normalized, but we omit that for simplicity.

\[ N = P_u \times P_v \]
\[ N' = P'_u \times P'_v \]
Bump mapping derivation

\[ N' = P'_u \times P'_v \]
\[ = \left( P_u + \frac{\partial B}{\partial u} N \right) \times \left( P_v + \frac{\partial B}{\partial v} N \right) \]
\[ = N + \frac{\partial B}{\partial u} N \times P_v + \frac{\partial B}{\partial v} P_u \times N \]
\[ = N + \frac{\partial B}{\partial u} N \times P_v - \frac{\partial B}{\partial v} N \times P_u \]

The “partial derivatives” are just differences between adjacent pixel values in the bump map.
Computing tangent space basis vectors

We now need to compute the vectors $P_u$ and $P_v$. These are the directions on the surface that correspond to zero change in the $v$ parameter and the $u$ parameter, respectively.
Computing tangent space basis vectors

\[ d_1 = p_1 - p_0 \]
\[ d_2 = p_2 - p_0 \]

\[ (s_1, t_1) = (u_1 - u_0, v_1 - v_0) \]
\[ (s_2, t_2) = (u_2 - u_0, v_2 - v_0) \]

\[ d_1 = s_1 P_u + t_1 P_v \]
\[ d_2 = s_2 P_u + t_2 P_v \]
Computing tangent space basis vectors

\[
d_1 = s_1 P_u + t_1 P_v \\
d_2 = s_2 P_u + t_2 P_v
\]

\[
\begin{pmatrix}
  d_1^T \\
  d_2^T
\end{pmatrix} = 
\begin{pmatrix}
  s_1 & t_1 \\
  s_2 & t_2
\end{pmatrix} 
\begin{pmatrix}
P_u^T \\
P_v^T
\end{pmatrix}
\]

\[
\begin{pmatrix}
P_u^T \\
P_v^T
\end{pmatrix} = \frac{1}{s_1 t_2 - s_2 t_1} 
\begin{pmatrix}
t_2 & -t_1 \\
  s_2 & s_1
\end{pmatrix} 
\begin{pmatrix}
d_1^T \\
  d_2^T
\end{pmatrix}
\]

We can now normalize these vectors and use them for bump mapping. To compute tangent vectors at a vertex, we average the vectors from the adjacent faces, as we did with normals.
Normal mapping

- Store the displaced normals directly. Reduces runtime overhead, at the expense of memory requirements.

- \((x,y,z)\) values in the tangent space are stored in the RGB channels. To compute the normal at a fragment, we simply multiply the (interpolated) tangent space basis by \((x,y,z)\)

Other Buffers
Accumulation buffer

• High-precision image buffer. Can integrate images that are rendered into the framebuffer. Supports anti-aliasing, motion blur, depth of field, soft shadows, etc.

• 16 bits for each red, green, blue, and alpha component: total of 64 bits per pixel.

• Supports the following operations:
  - Clear: set all values to zero.
  - Add with weight: Each pixel in the drawing buffer is added to the accumulation buffer after being multiplied by a floating-point weight that can be positive or negative.
  - Return with scale: The contents of the accumulation buffer are returned to the drawing buffer after being scaled by a positive floating-point constant

• Can integrate up to 256 images without loss of prevision, and even more using weight less than 1.0
Accumulation buffer

motion blur

depth of field

soft shadows
Accumulation buffer

motion blur

depth of field

soft shadows

Haeberli and Akeley, SIGGRAPH 1990
Stencil buffer

- Integer buffer, 8 bits per pixel. Can be used to restrict rendering to parts of the screen.
- Implements state machine at every pixel. A *stencil test* determines whether or not a fragment is written.
- Application 1: portals and mirrors
- Application 2: a different technique for simulating shadows, called *shadow volumes*