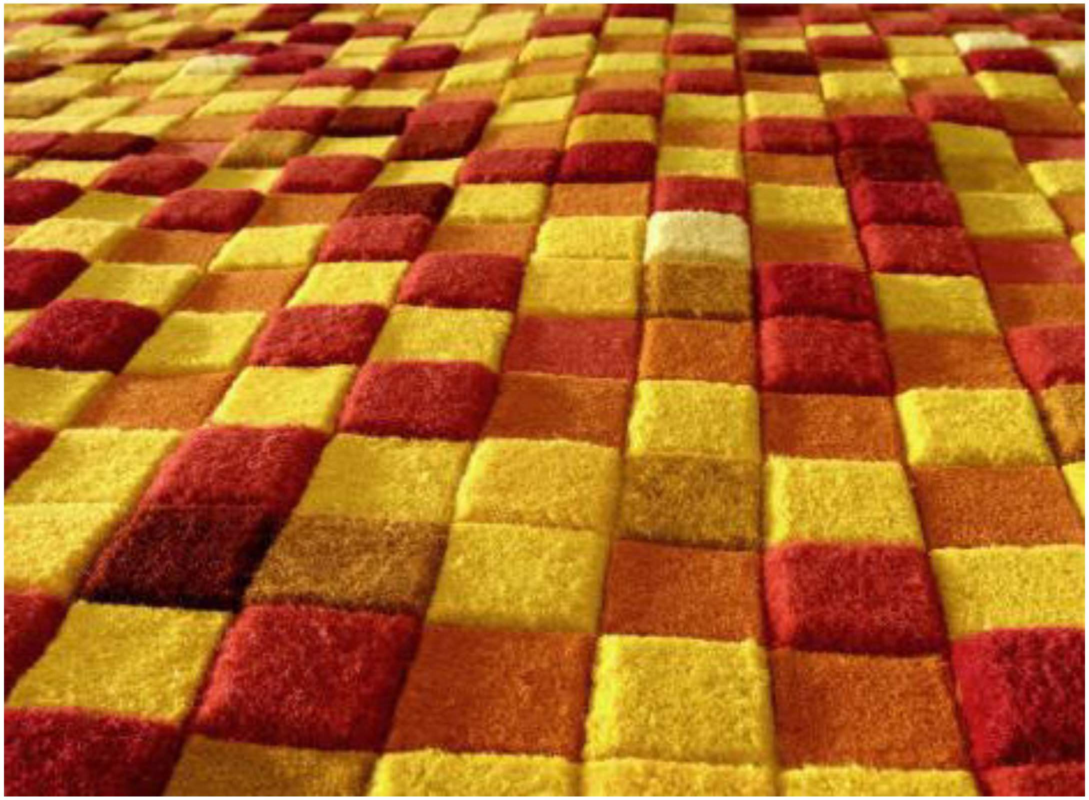
Texture Mapping

CS384G – Fall 2012

Christian Miller

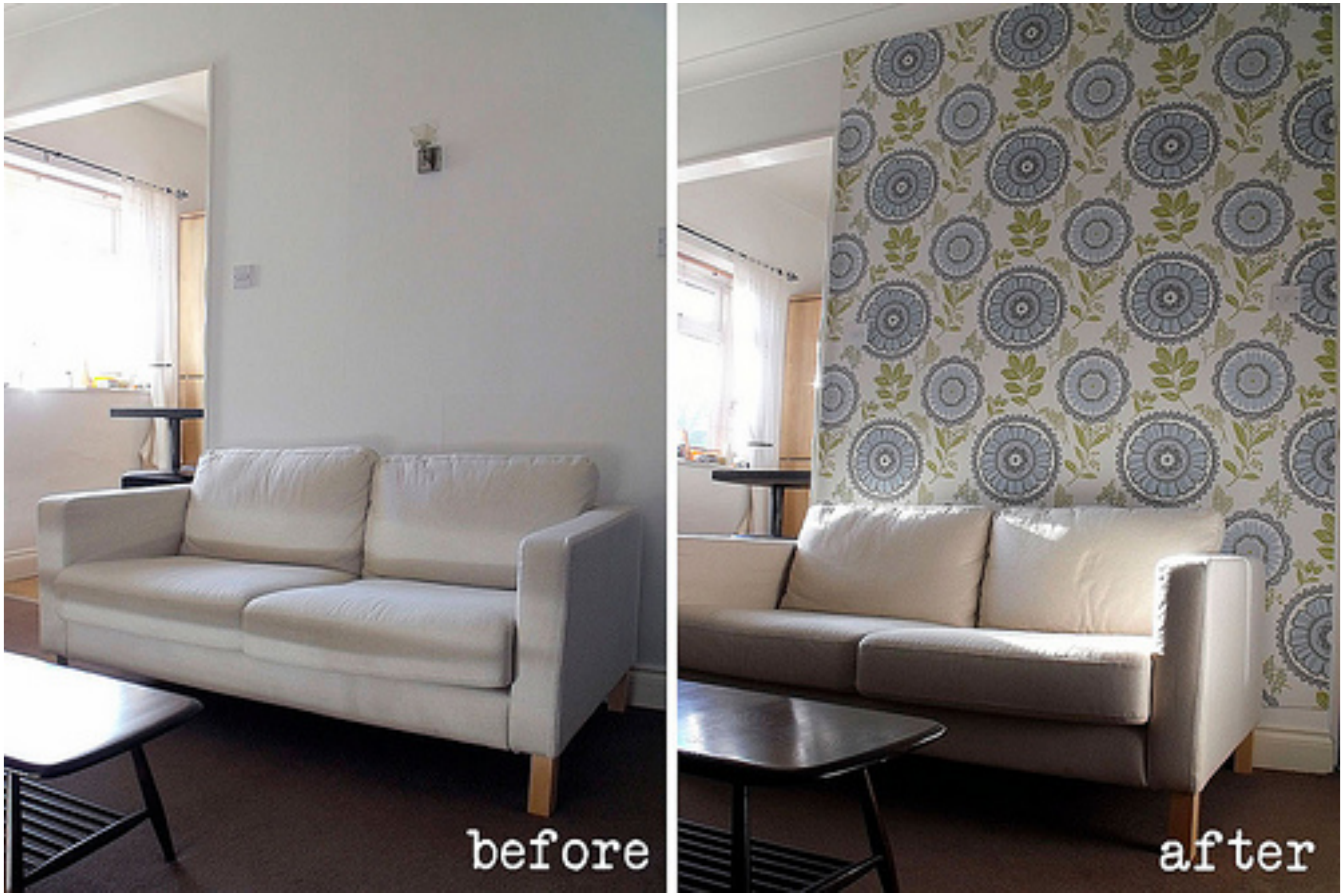
Surface detail



Surface detail

- Most things have a lot of detail, and simple polygons or triangle meshes are poor approximations
- Modeling all that detail with simple primitives would take eons, and enormous amounts of storage
- Rendering all of it would take forever too
- We can’t just give up, so we need some way to make surfaces look more detailed than they actually are…

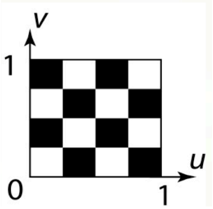
Wallpaper



Texture mapping

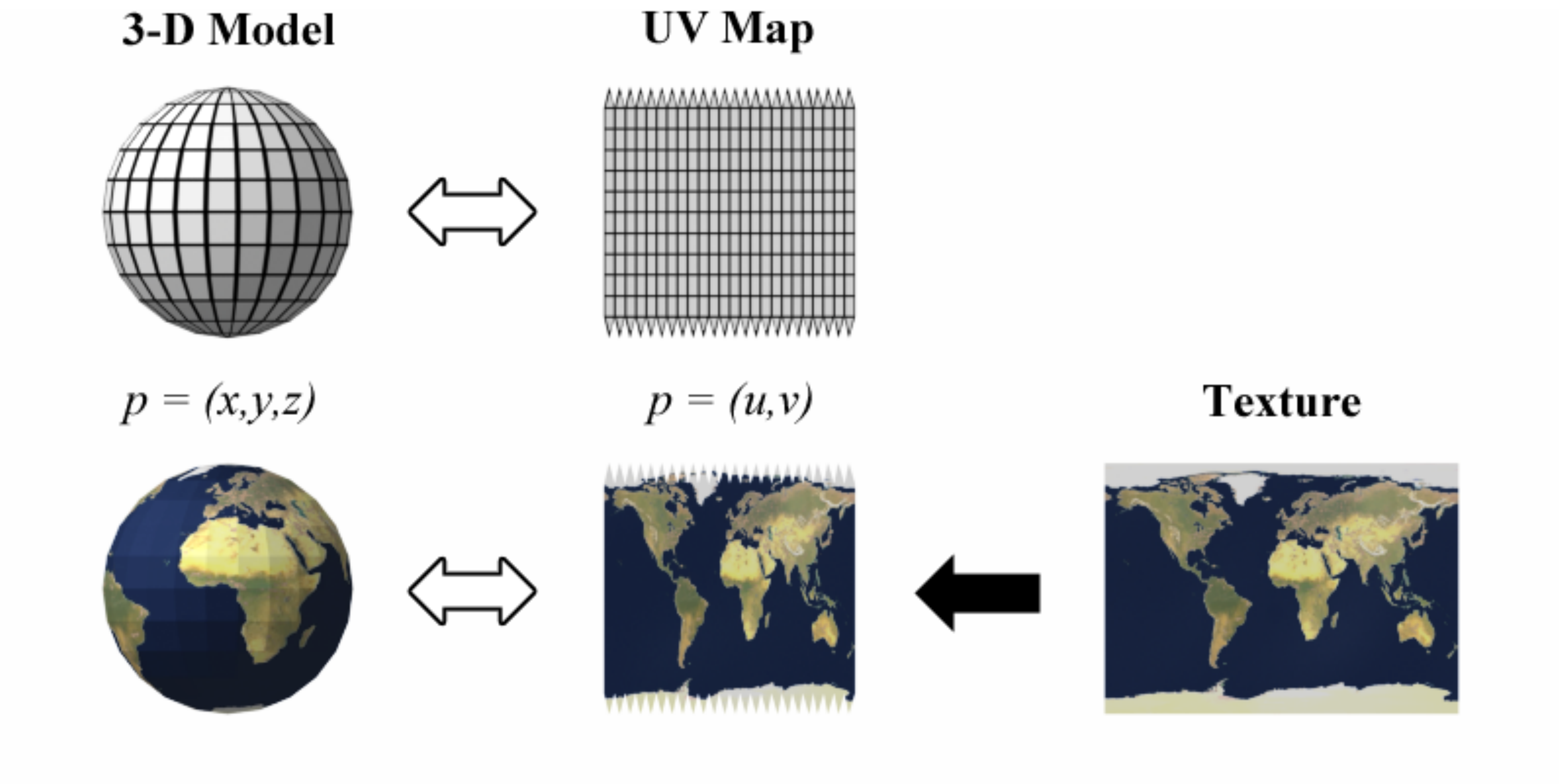
- Take an image with the surface detail on it
  - The pixels that make up the texture are often called texels
- Stretch it over the surface
- When rendering a pixel, look up the diffuse color from the texture and use the rest of the light model as usual

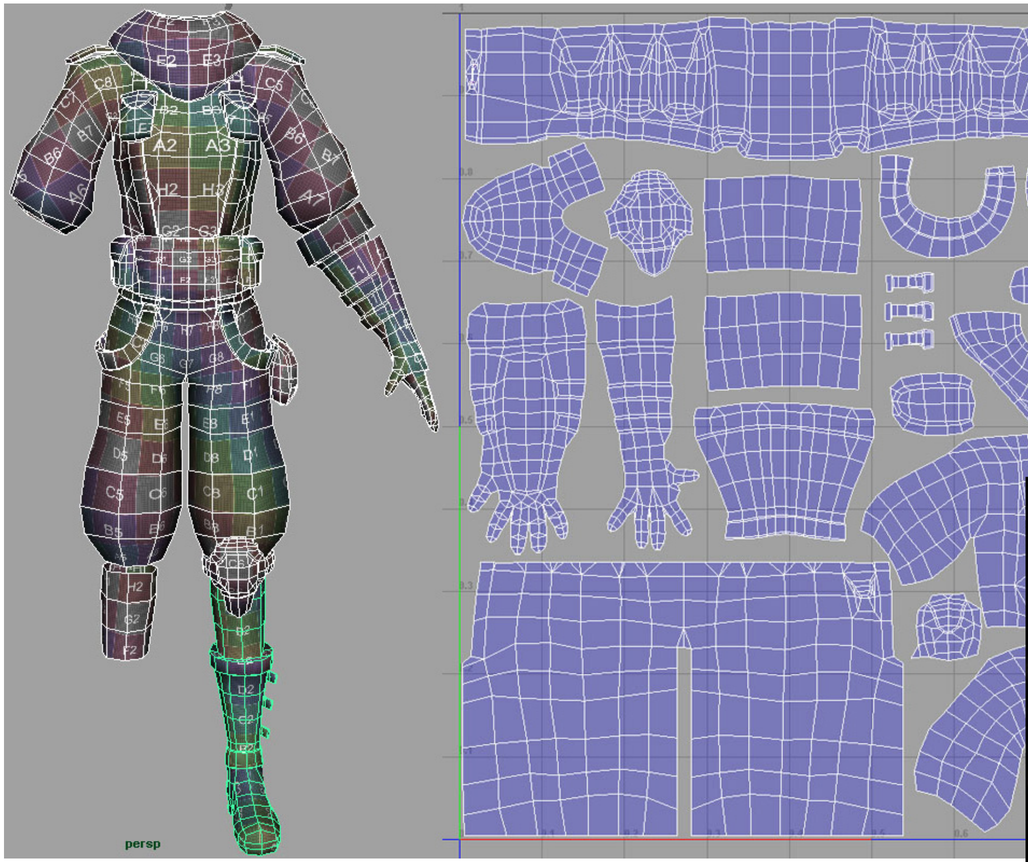
Mapping to a surface

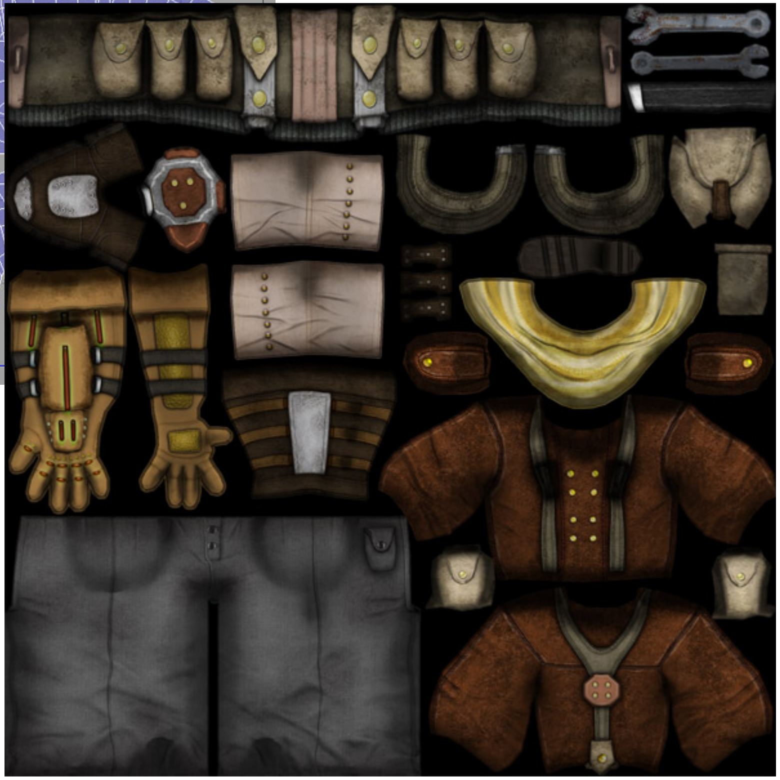


- Accomplished through texture coordinates (u, v)
- The texture image has coordinates (0, 0) in the lower left corner and (1, 1) in the upper right
- For a mesh, have user specify the (u, v) coordinates at each vertex
- To render a pixel, interpolate (u, v) at the intersection point, use those texcoords to look up the right color from the texture

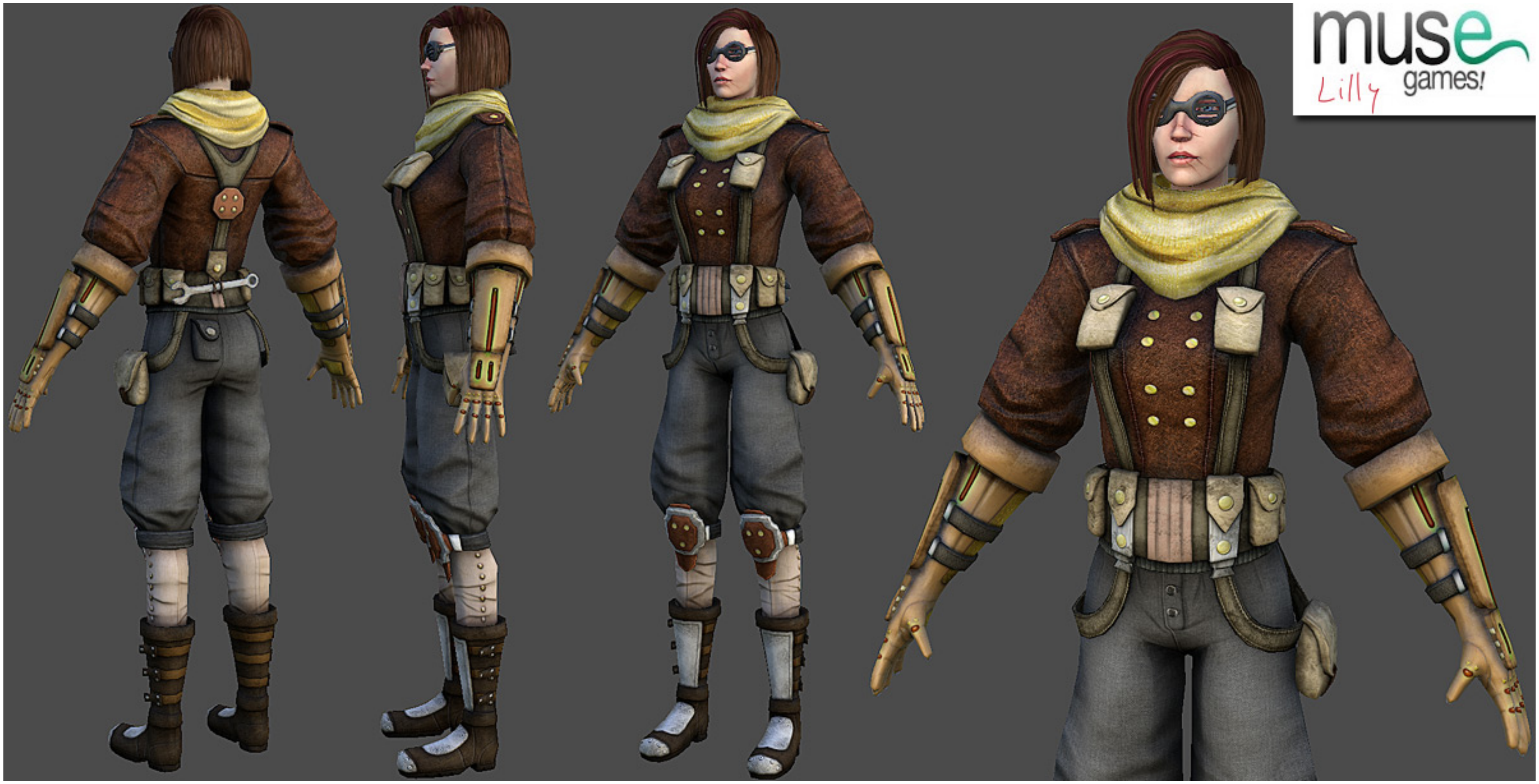
Specifying texcoords





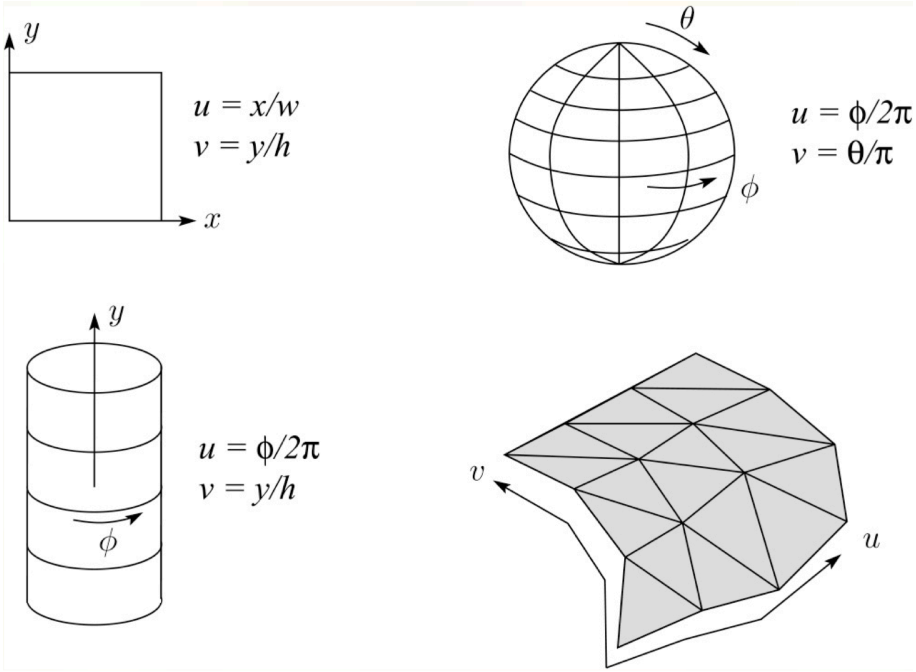


[Muse Games]

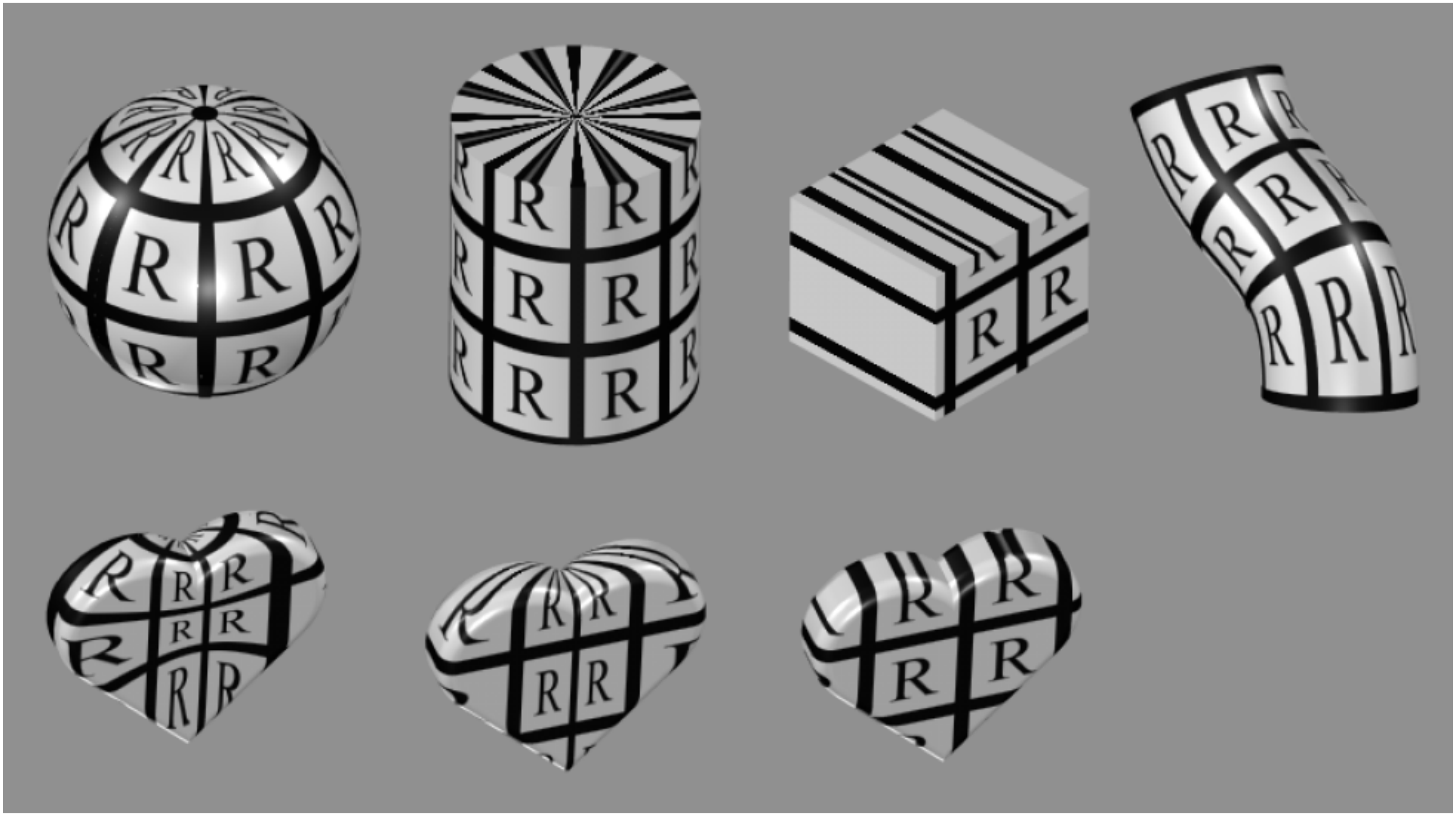


[Muse Games]

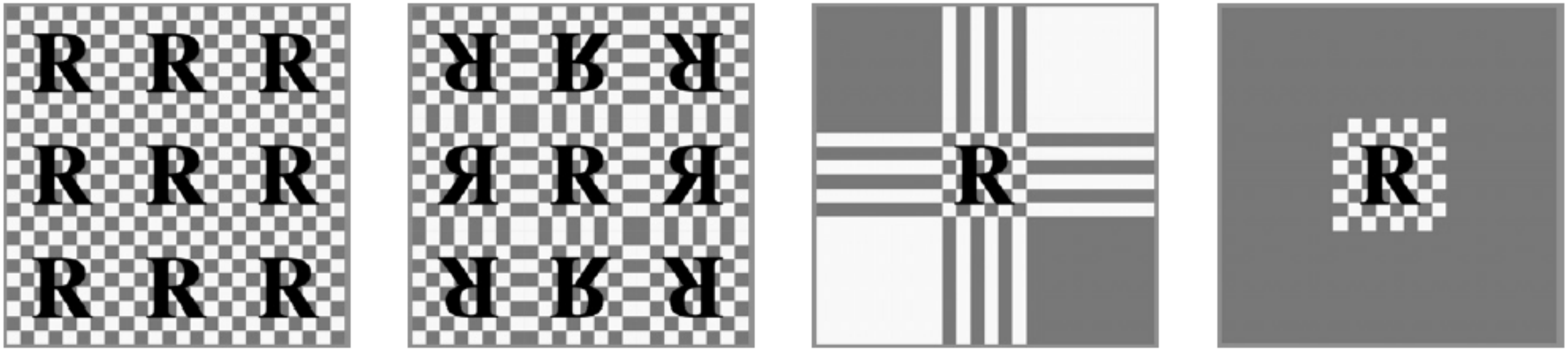
Alternate ways of generating UVs



- You don’t necessarily need to specify texcoords on a per-vertex basis
- Sometimes a simpler function can do it automatically for you

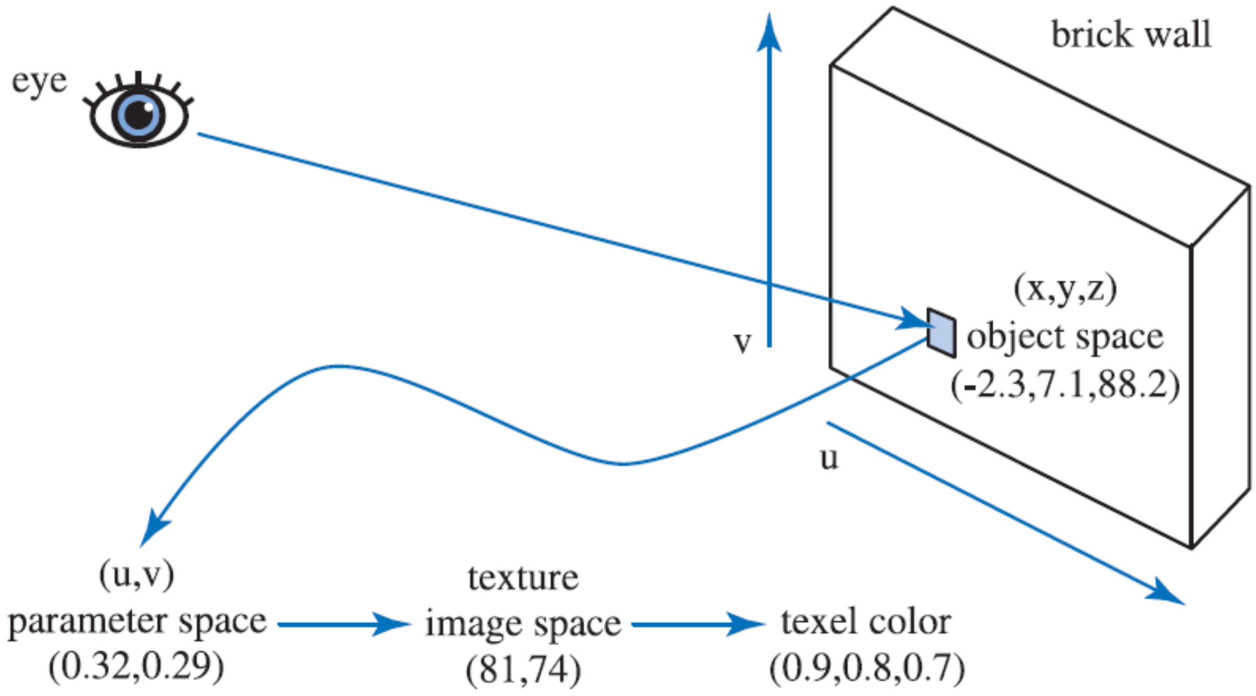


Texture edge modes

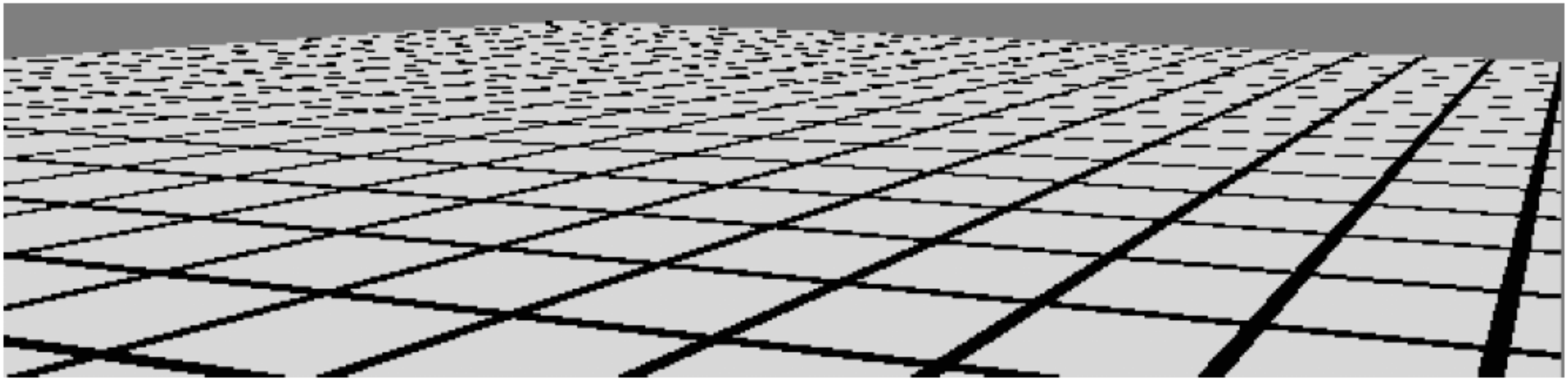


- What do you do when you get a texcoord outside [0, 1]?
- Adopt some convention:
  - Loop around and start at the other side (wrapping)
  - Reflect the image backwards (mirroring)
  - Repeat the edge pixels (clamping)
  - Default to some other color (bordering)

Texture lookup



Texture filtering

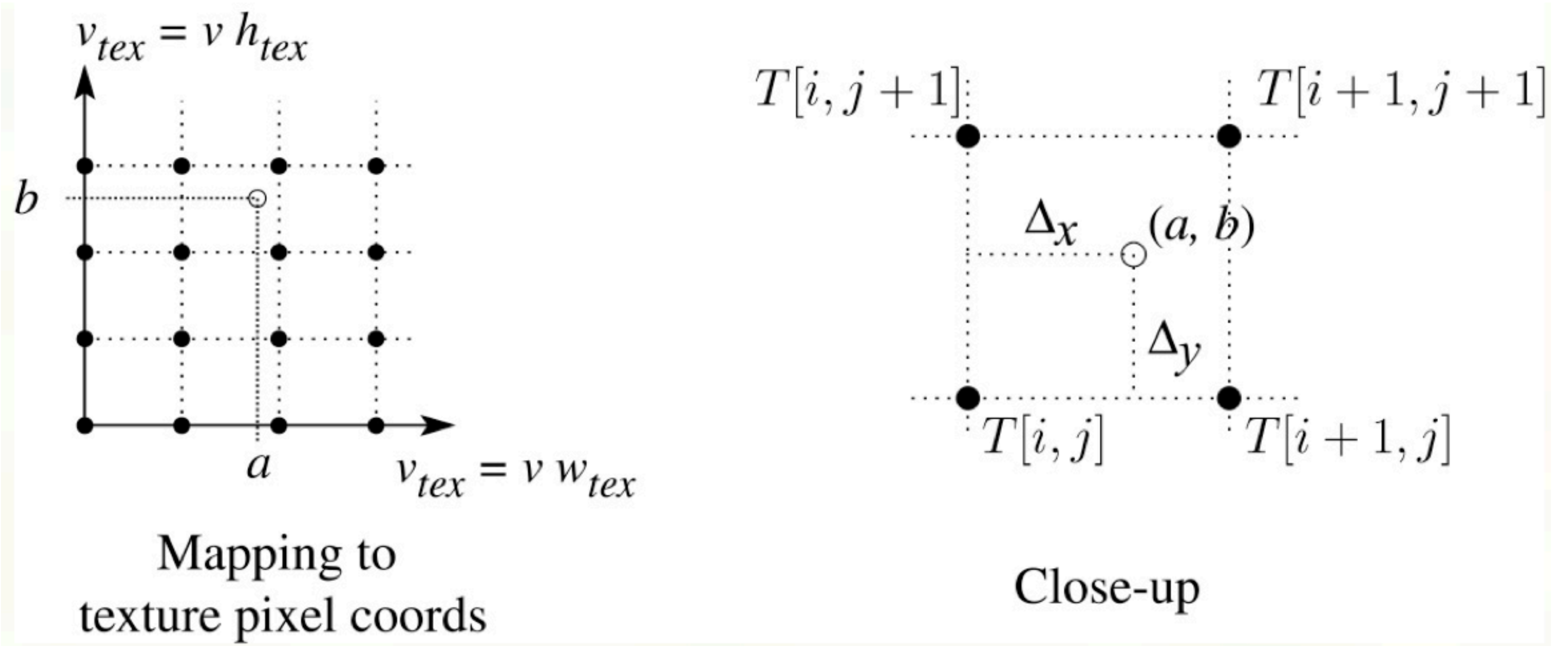


- Simply returning the pixel you hit in the texture(nearest neighbor) looks terrible
- Far away, the texture is undersampled and unrecognizable
- Up close, the texels look huge and blocky

Magnification

- The image looks really blocky, since each texel covers several pixels
- Since we have more pixels covering the area than there are texels, we need to fake data that isn't there
- Blurring the image is a good idea, since even that looks better than giant sharp-edged texels
- The usual fix is bilinear interpolation (bilerp)

Bilinear interpolation

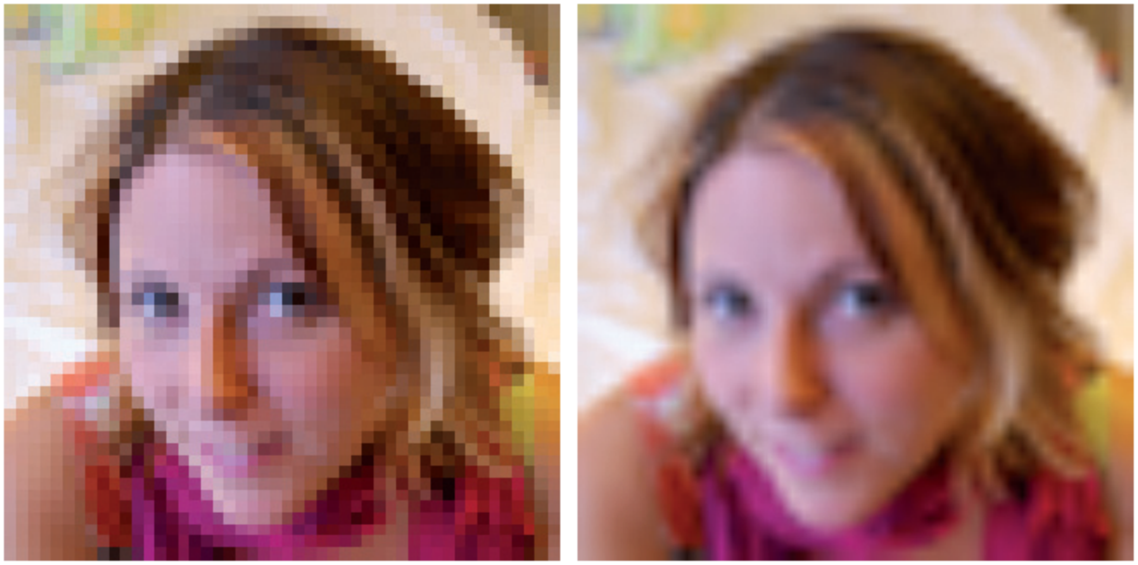


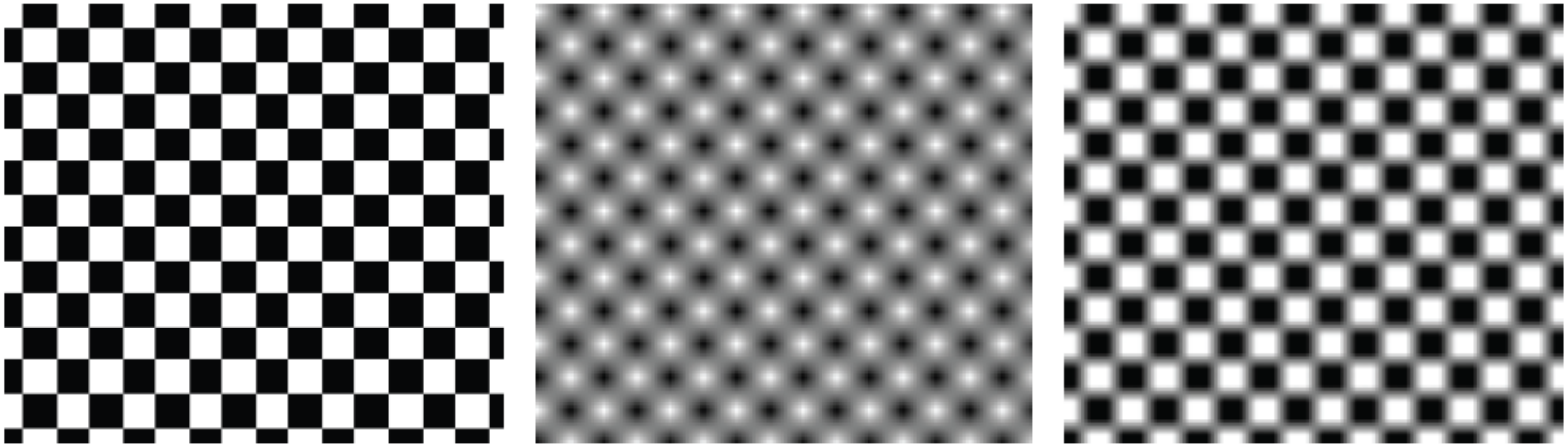
Mapping to texture pixel coords

Close-up

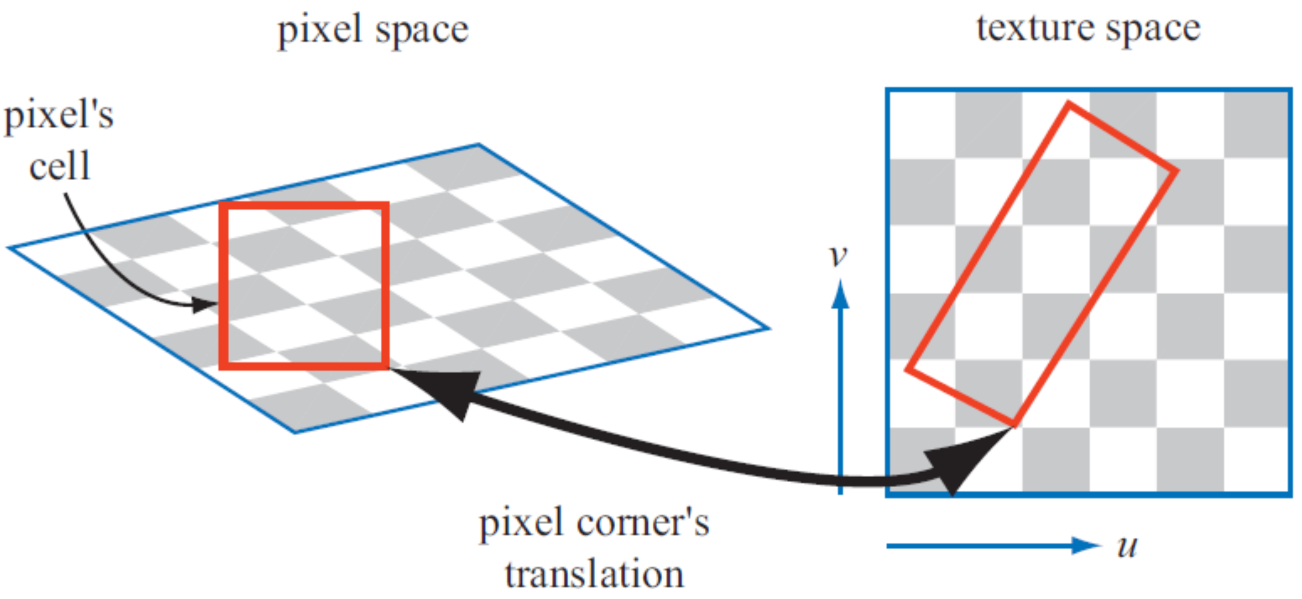
- Sample neighboring texels, blend them linearly
- T(a, b) = (1-Δx)(1-Δy) T(i, j) + Δx (1-Δy) T(i+1, j) + (1-Δx)Δy T(i, j+1) + Δx Δy T(i+1, j+1)

Bilerp results





Minification



- One pixel on the screen can cover any number of texels
- Coverage area in texture space is an arbitrary shape
- This is an undersampling issue, which means it can be addressed with anti-aliasing methods

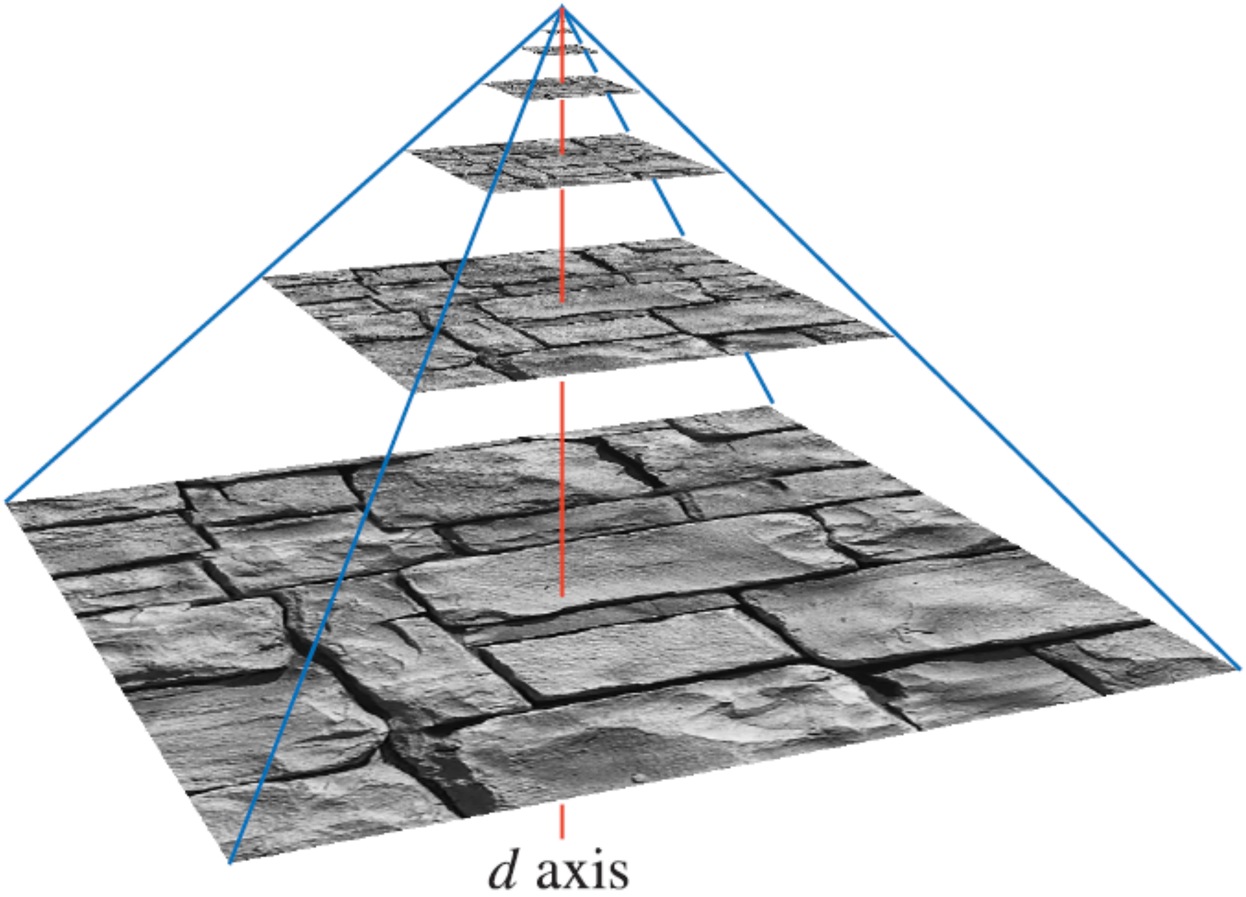
Supersampling

- Since we have several texels being covered by a single pixel, the analytic method is to take an average of all texels weighted by their intersection area with the pixel
  - This is expensive and complicated
- Can be approximated by sending several jittered rays through the area of the pixel and averaging them
  - Still expensive, but not complicated
  - Most high-quality renders do this anyway, since it smooths out jaggies on edges as well as textures
  - Used very commonly in raytracers

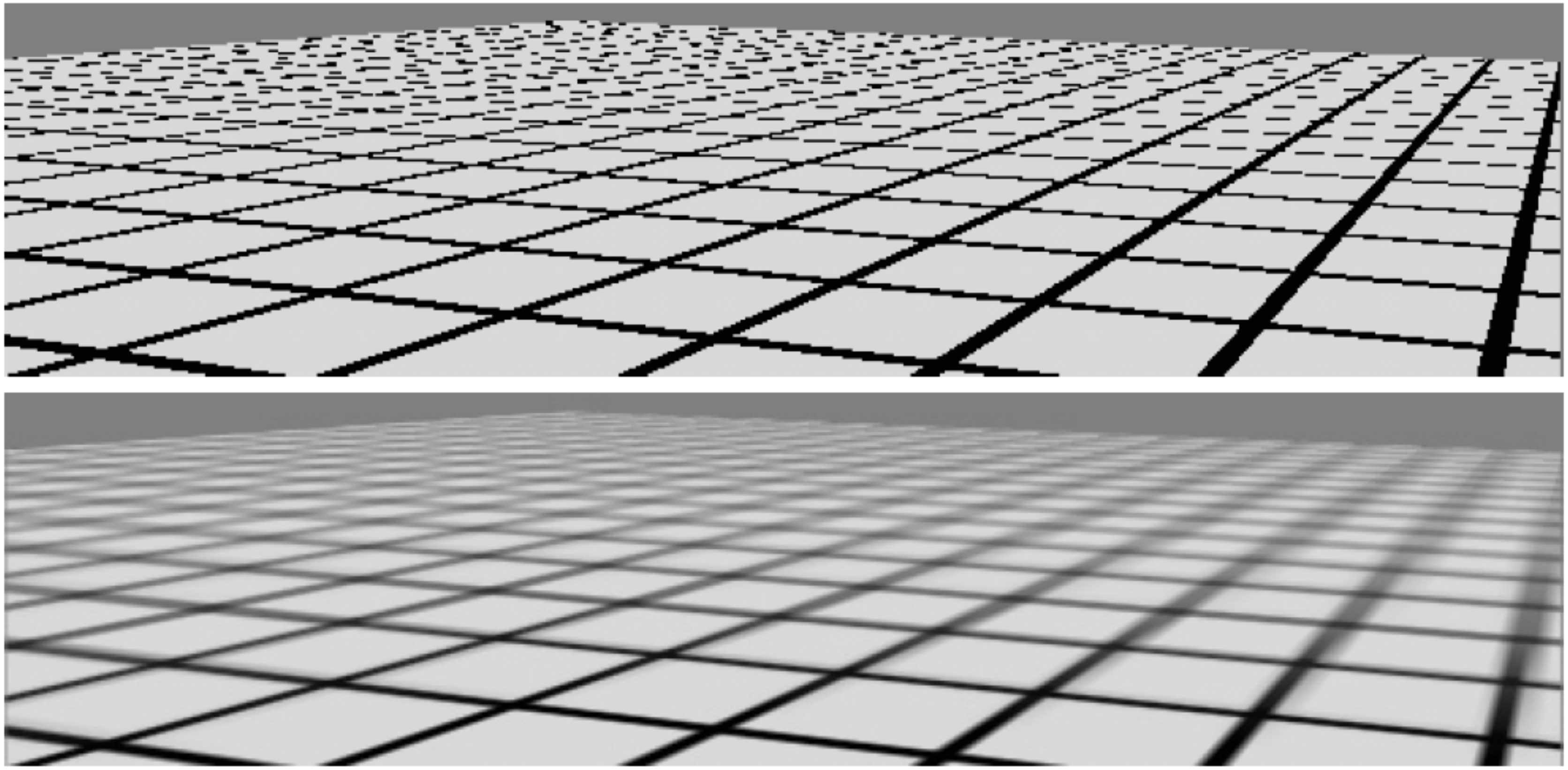
Mipmapping

- In a realtime system, you may not be able to afford taking tons of samples per pixel
- Instead, take the original textures and make several pre-blurred versions of them
  - Each texture is half the size of the larger one, giving a pyramid of textures from each full image
  - Requires 1/3rd more memory than just the original texture
- Then at runtime, use distance from camera and surface angle to pick a version of the texture to sample

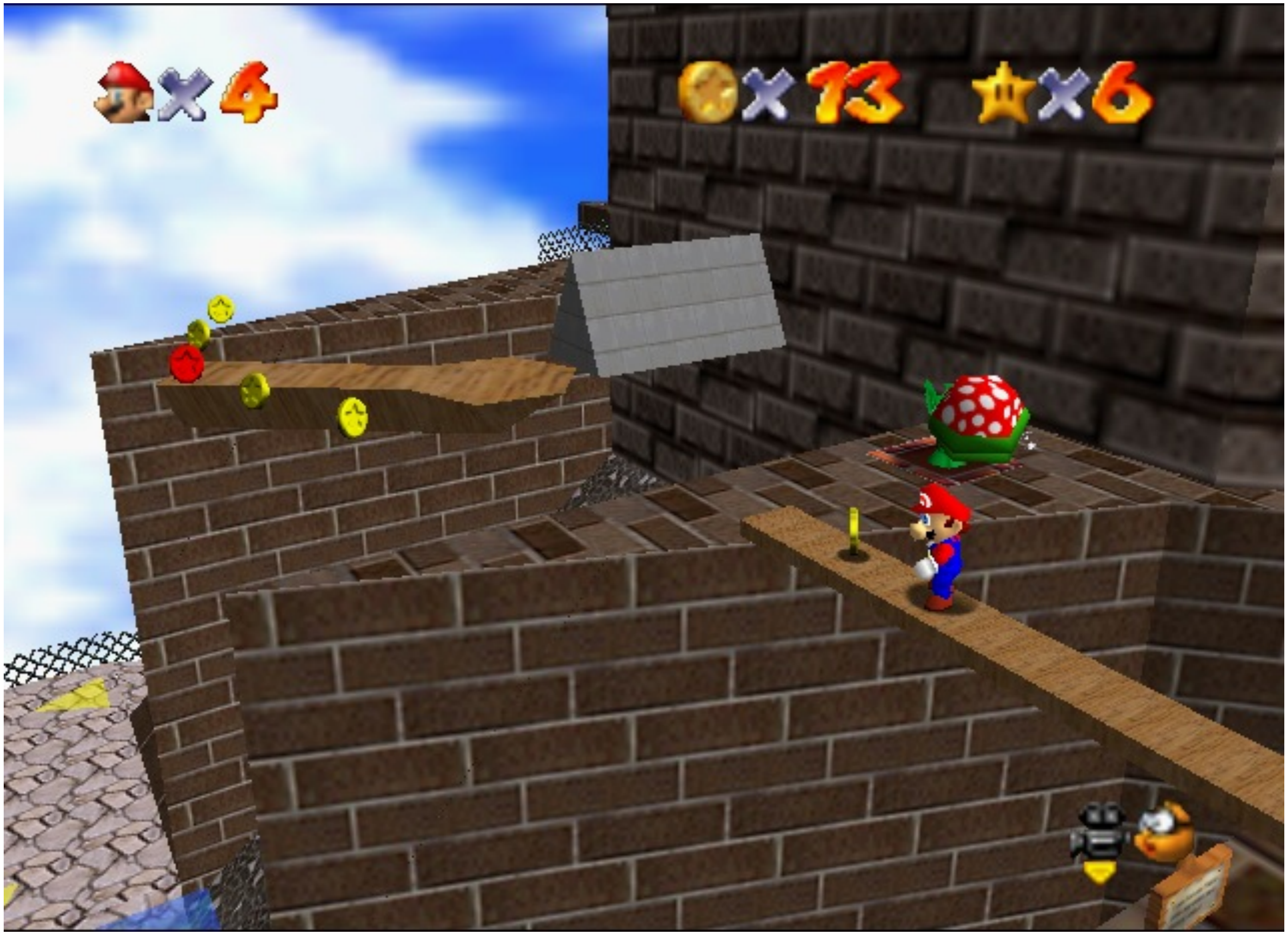
Mipmap pyramid



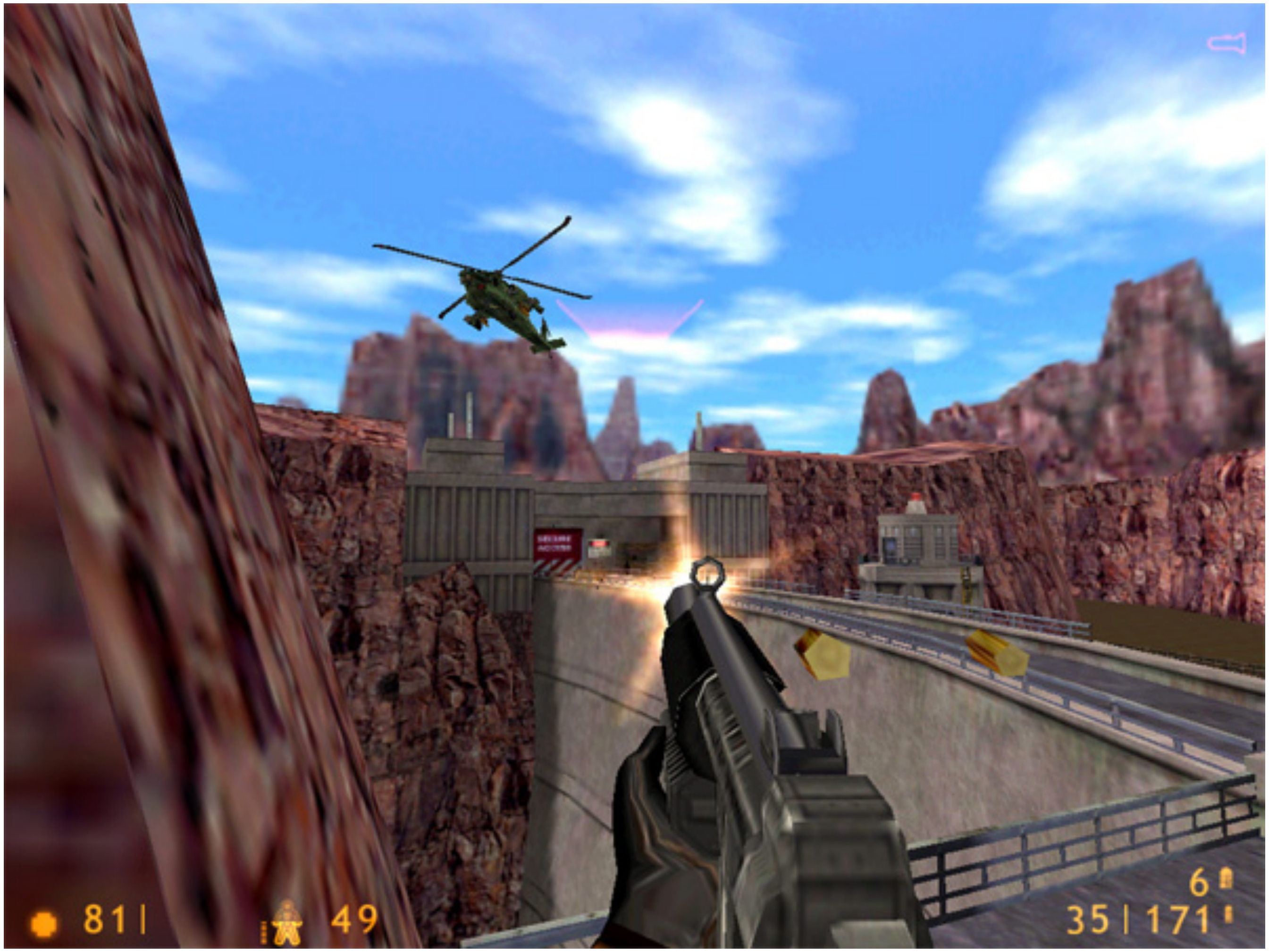
Mipmap results



Standard texture mapping



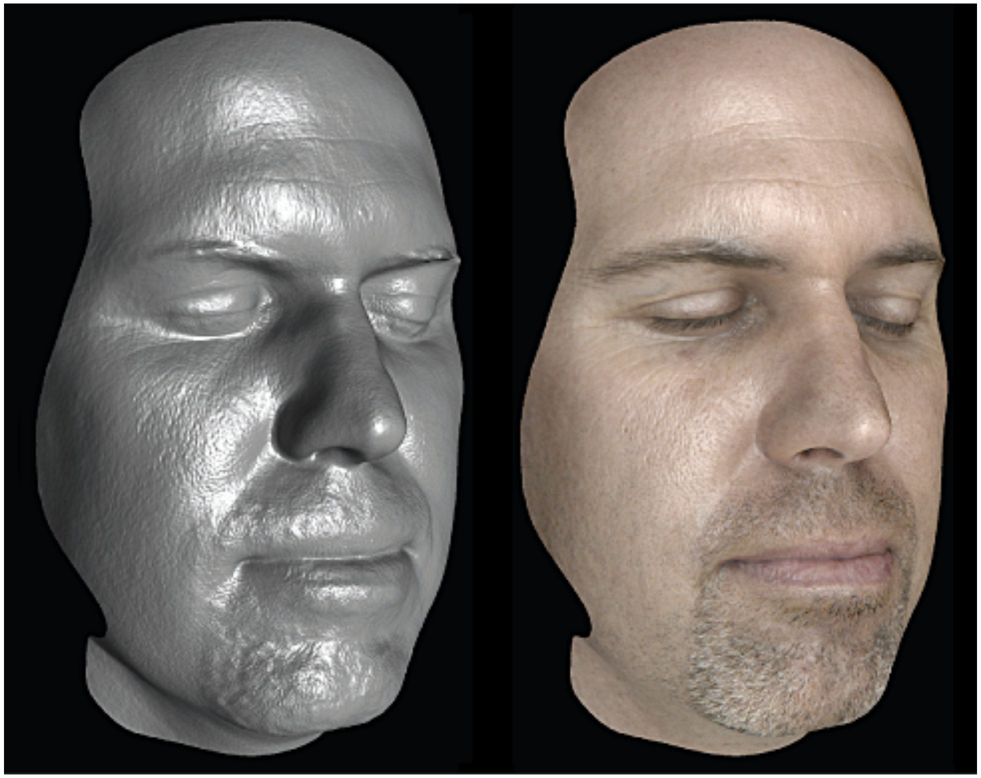
Standard texture mapping



Other maps

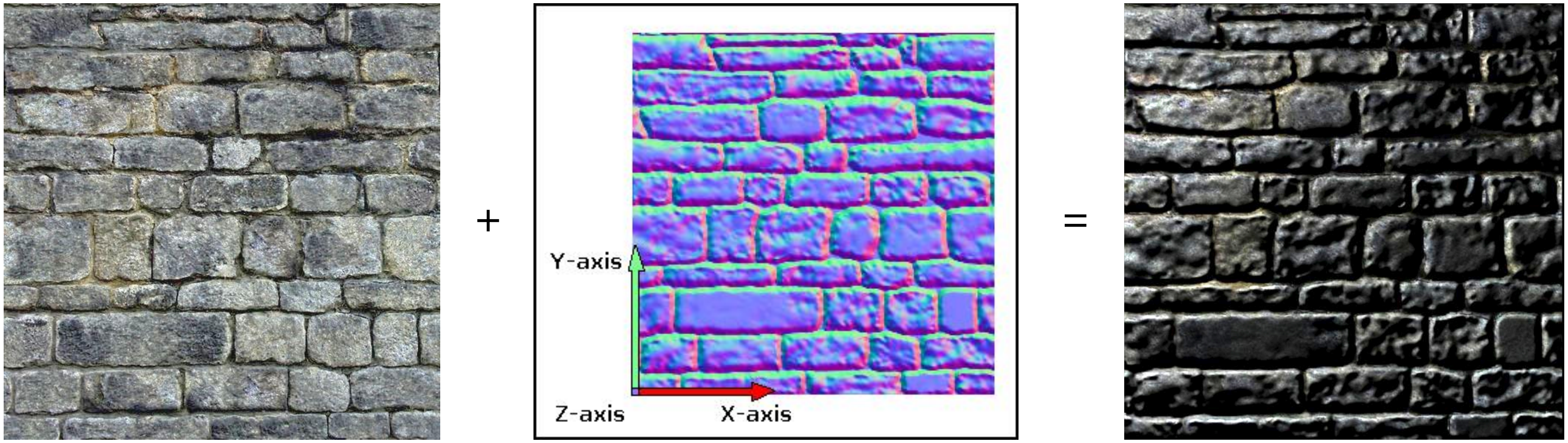
- So far, we’ve just been using texture maps to alter the diffuse component of the lighting model
- In the most general case, a texture just represents some function attached to a surface
- What happens if we use it to store other components of the lighting model?

Specular mapping



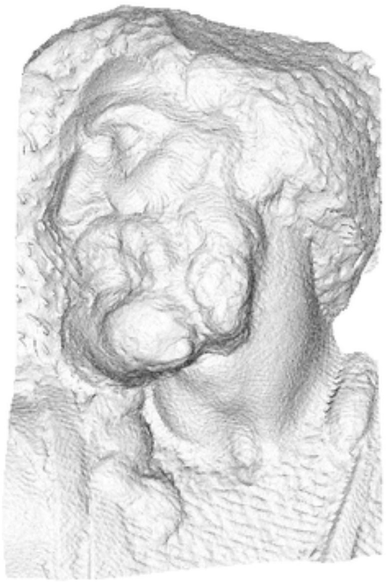
- Use a texture to store the intensity of the specular term

Normal mapping

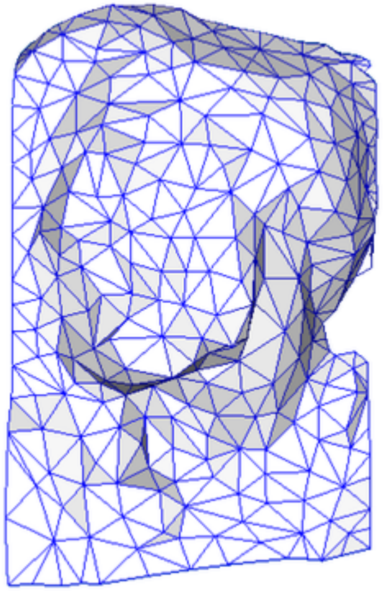


- Store surface normal (relative to geometry) compressed in a texture map
- At runtime, look up normal in texture and add it to the usual normal that's interpolated from the vertices
- Makes a huge visual difference without adding geometry

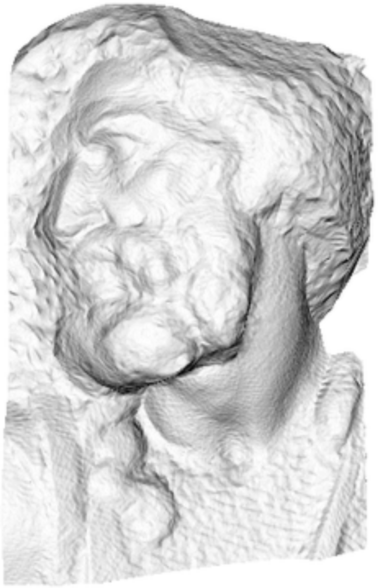
Normal mapping



original mesh
4M triangles

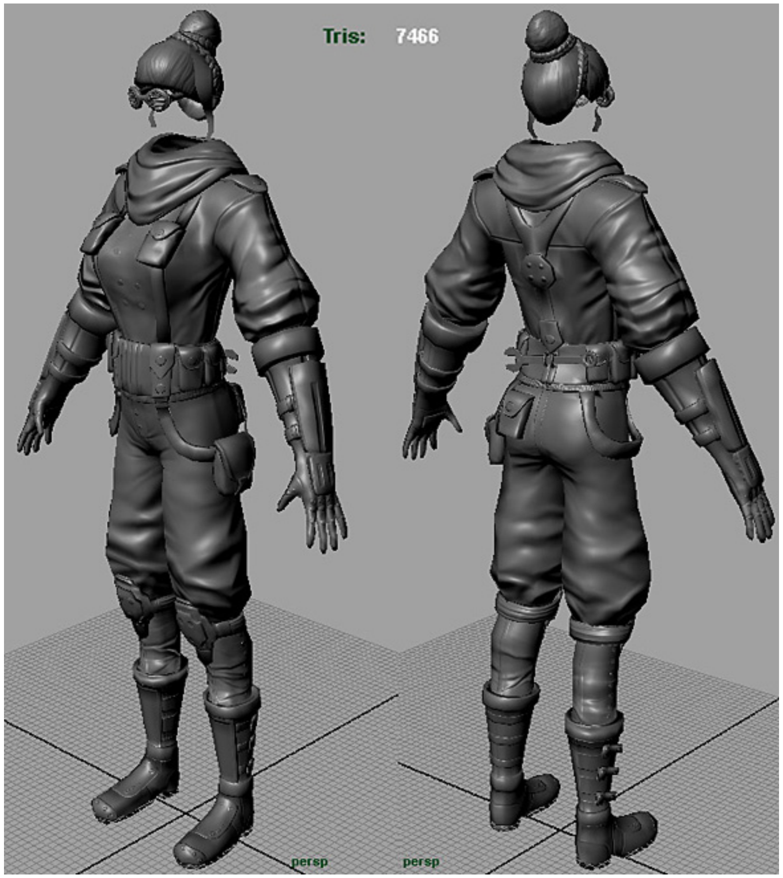


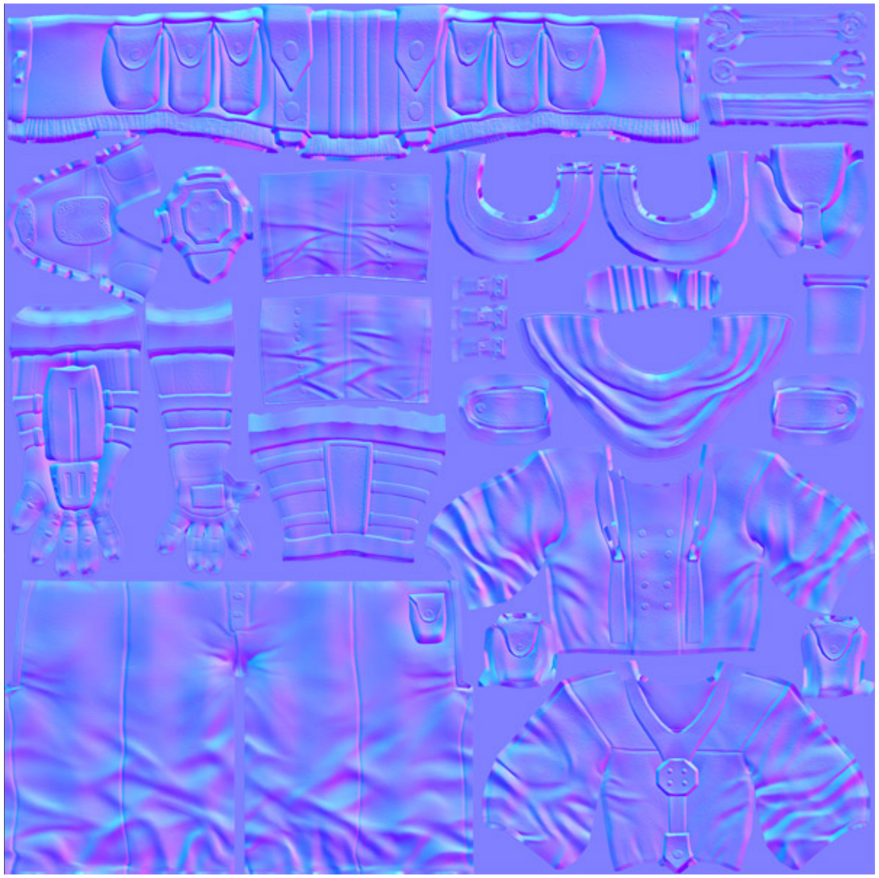
simplified mesh
500 triangles



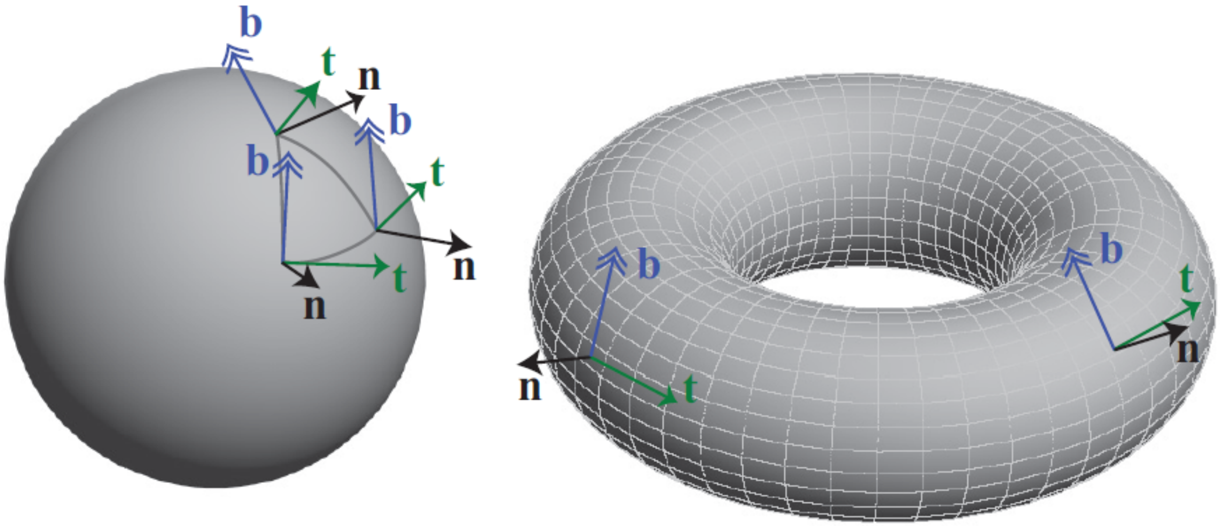
simplified mesh
and normal mapping
500 triangles

Normal mapping





Normal mapping details

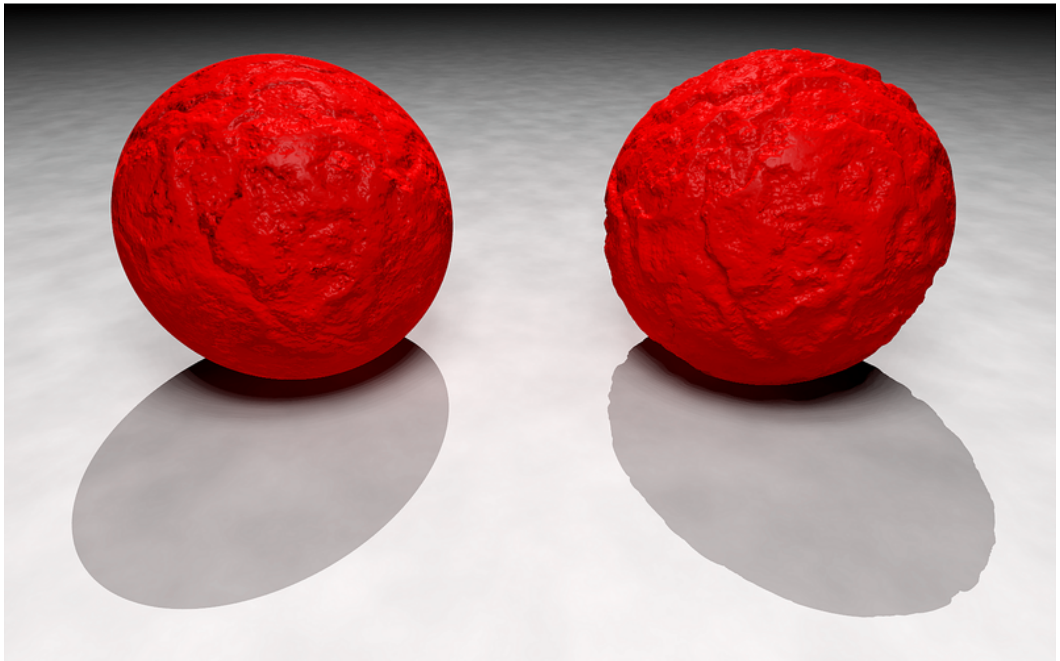


- Normals are stored in texture by making the (r, g, b) components the (x, y, z) values of the normal

- Z is clearly in the normal direction from the surface, but the X and Y directions are unspecified

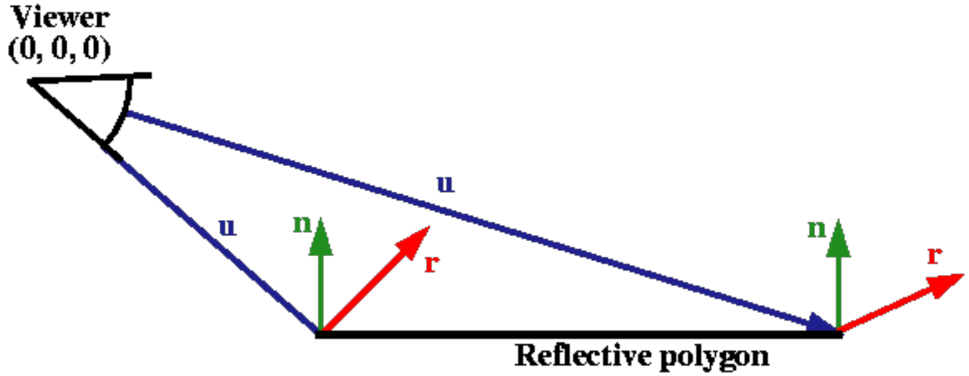
- Need to add tangent and binormal vectors to form a full coordinate system at every point

Displacement mapping



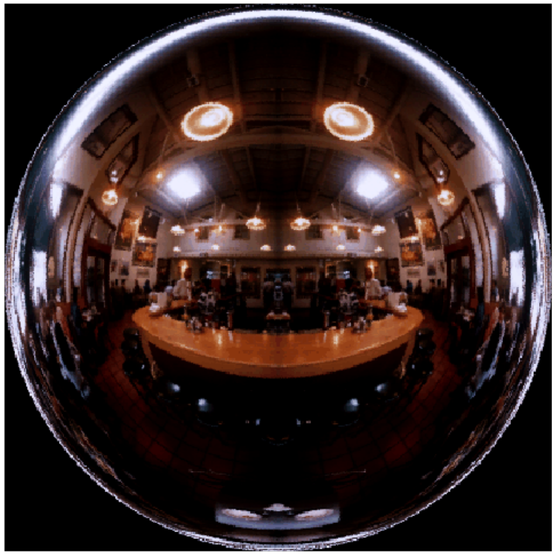
- Normal maps don't add any detail to the silhouette of an object, since the actual geometry is still simple
- You can finely subdivide the geometry and use a displacement map to offset the individual vertices

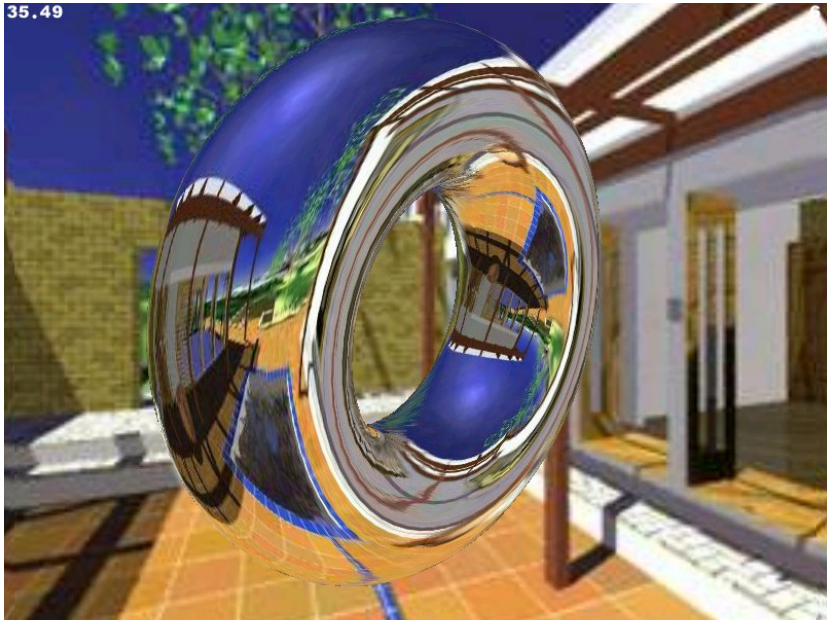
Environment mapping



- There's no reason a texture needs to be glued to a surface, we can compute texcoords on the fly
- For example, if we're running in realtime and want reflections but can't afford to cast rays…
  - Store the surrounding environment in a texture map
  - Compute reflection vectors at each vertex, use those to set texture coordinates, then the environment gets mapped onto the surface as if it was reflected

Spherical environment mapping



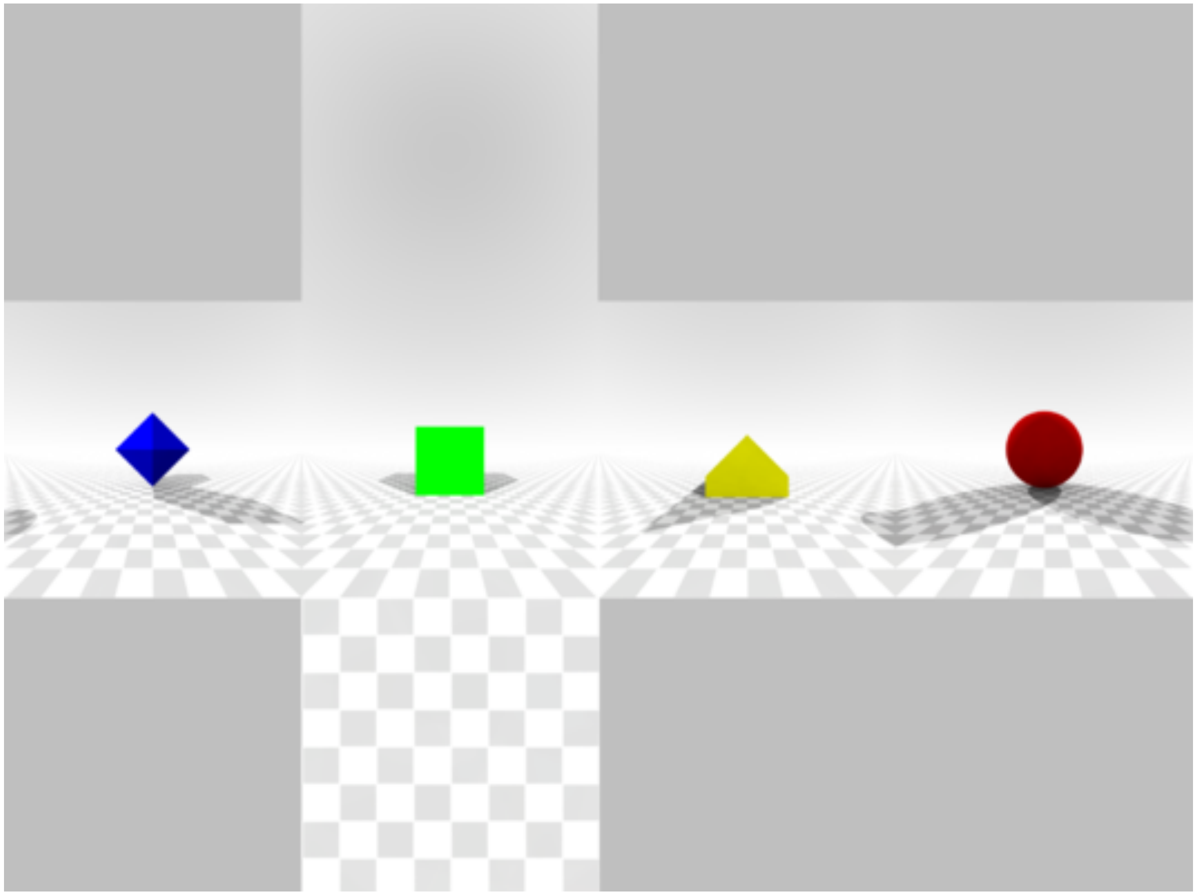


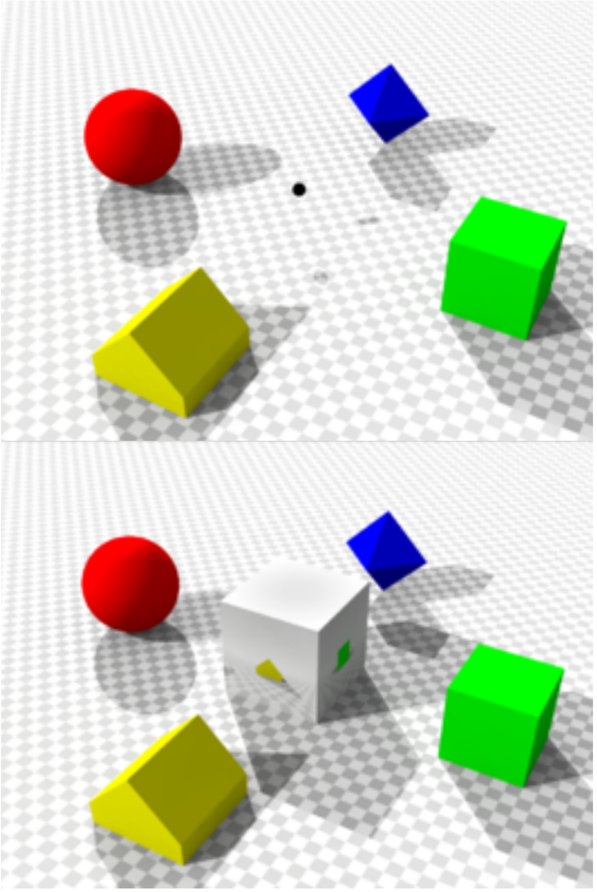
- Store the environment as a picture of a perfectly reflective sphere, easy math to compute texcoords
- Only covers a hemisphere, reflections remain fixed relative to the camera

Cube mapping

- Spherical environment mapping can’t get all sides of an object, so you can use texture maps on the sides of a cube instead
- Math is more complicated, but results are better
- Easier to render environment map at runtime too

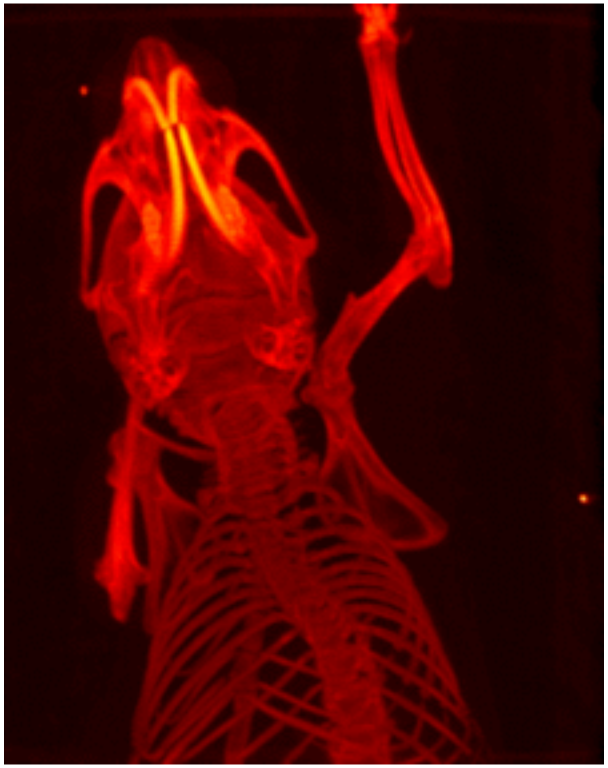
Cube mapping





Cube mapping

3D textures



It’s a big iceberg

- There are enormous numbers of ways that texture mapping has been used
- Basically every cool graphics effect you see in realtime is a texturing trick
- A good chunk of prerendered stuff is too