159.709 Computer Graphics

Skybox

Daniel Playne
d.p.playne@massey.ac.nz

Skybox

A skybox is a technique used to make a 3D scene appear as though it is connected to a larger world without the prohibitive cost of rendering an entire world.

Skyboxes generally show very distant objects such as:

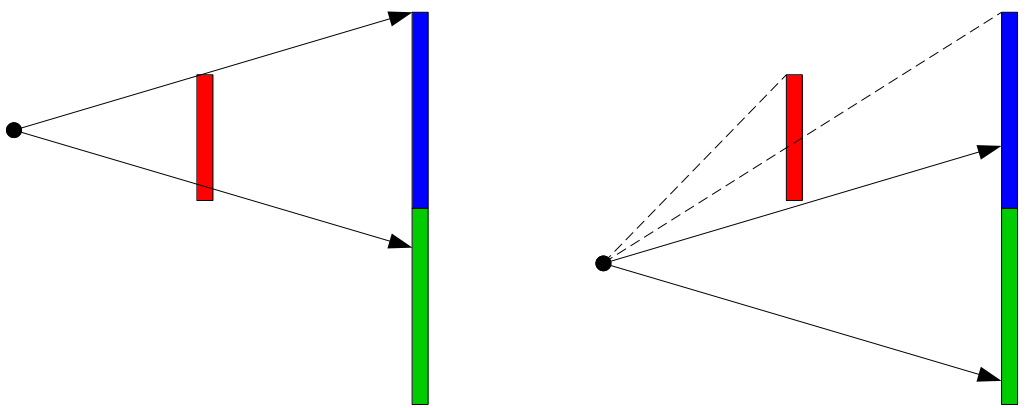
- Mountains
- Skyscrapers
- and of course the sky

Parallax Effect

Parallax is the displacement of the apparent position of an object when viewed from different positions.

The effect can be easily seen by holding something in front of you and moving your head from side to side.

The parallax effect is significant for objects that are close to the viewer and is one of the methods humans use to perceive depth.



Parallax Effect

For objects that are sufficiently far away, there is almost no parallax effect when the viewer moves.

The appearance of a distant mountain does not change significantly when the viewer moves by one metre or ten metres or even a kilometre.

The distant objects may be completely or partially obscured by closer objects but their actual appearance does not change.

Skybox

Skyboxes take advantage of this fact to represent the incoming light from the surrounding distant environment as a texture map.

These maps can be used to directly render the distant objects in the surrounding environment that encompasses the scene.

These distant objects should be far enough away that movement throughout the 3D scene does not affect the appearance of the objects in the skybox.

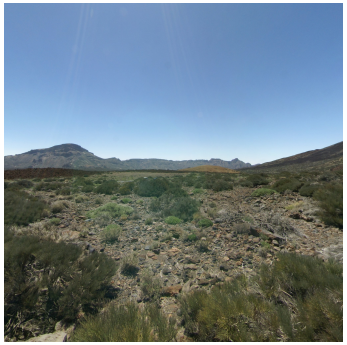
Cube Map

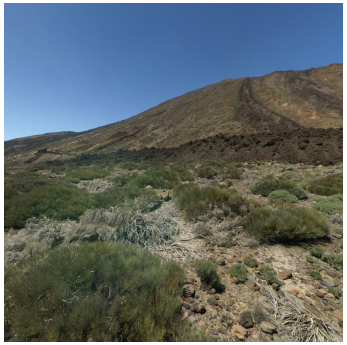
A common way to represent a skbyox is to use a ***cubemap*** which is a set of six textures (sometimes combined together into a single image) that represents the view of the environment in six different directions $(+x, -x, +y, -y, +z, -z)$.

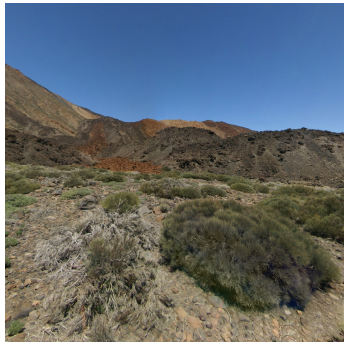
The skybox can be drawn around the viewer using these textures applied to a mesh.

For simplicity, the mesh is usually a cube (but does not necessarily have to be).

Cube Map







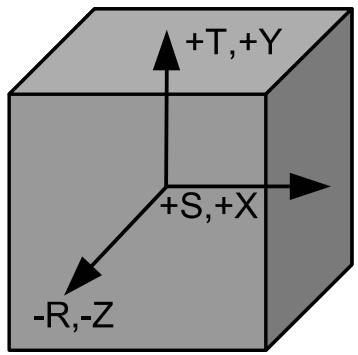
Cube Map

The six textures of the cube map are loaded to six different texture targets which represent one side of the cube each.

- `GL_TEXTURE_CUBE_MAP_POSITIVE_X`
- `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`
- `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`
- `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`
- `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`
- `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`

Cube Map

A cube map is not accessed through two-dimensional UV coordinates (or properly ST coordinates in OpenGL). Instead they are accessed via three-dimensional vectors (STR coordinates).



The cube face (texture) to access is chosen by the component (S,T,R) with the largest magnitude. The other two coordinates are then used to select a texel from that face.

Cube Map

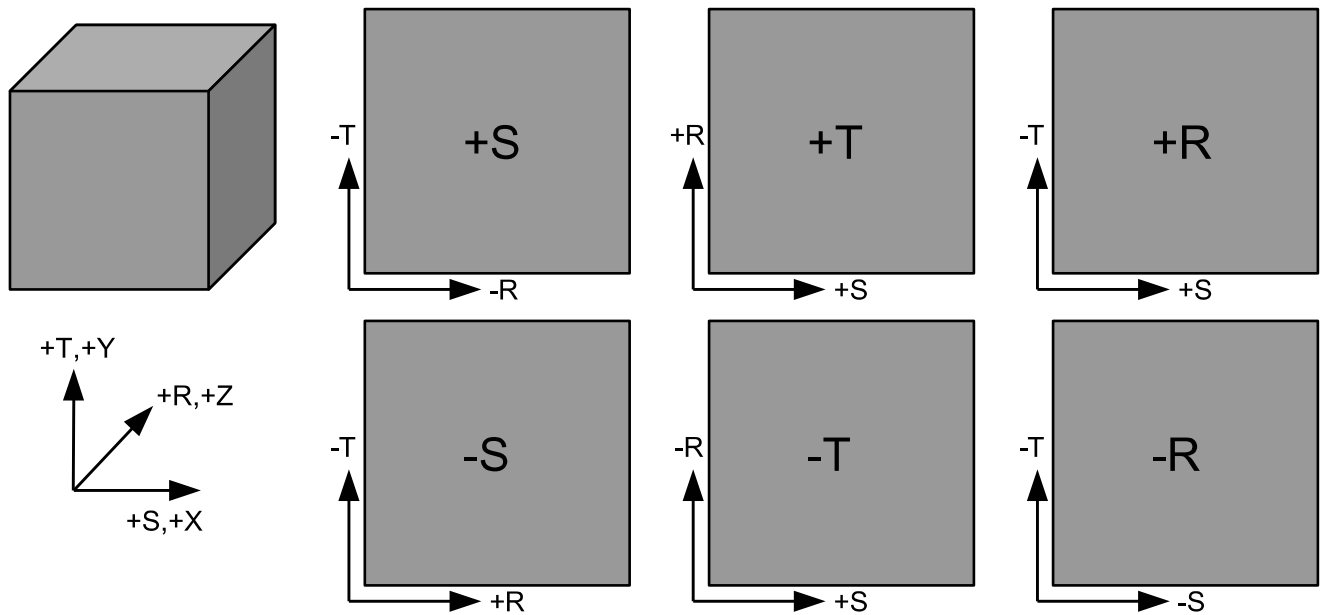
The coordinates for cube maps are not always so simple to understand, the following table describes how the three-dimensional (vx,vy,vz) vector is mapping into specific texture coordinates (st):

major axis	target	sc	tc	ma
+vx	TEXTURE_CUBE_MAP_POSITIVE_X	-vz	-vy	vx
-vx	TEXTURE_CUBE_MAP_NEGATIVE_X	+vz	-vy	vx
+vy	TEXTURE_CUBE_MAP_POSITIVE_Y	+vx	+vz	vy
-vy	TEXTURE_CUBE_MAP_NEGATIVE_Y	+vx	-vz	vy
+vz	TEXTURE_CUBE_MAP_POSITIVE_Z	+vx	-vy	vz
-vz	TEXTURE_CUBE_MAP_NEGATIVE_Z	-vx	-vy	vz

```
s = ( sc / |ma| + 1 ) / 2
t = ( tc / |ma| + 1 ) / 2
```

Cube Map

Which may be easier to understand in the following image (note the handedness of the axes):



Loading a Cube Map

```
1  // Load a CubeMap Texture from file
2  GLuint loadTextureCubeMap(const char *filename[6], int &width,
3                            int &height, int &n) {
4     // Generate and bind texture
5     GLuint texture;
6     glGenTextures(1, &texture);
7     glBindTexture(GL_TEXTURE_CUBE_MAP, texture);
8
9     // Load six faces
10    for(int i = 0; i < 6; i++) {
11       // Load image from file
12       unsigned char *image = loadImage(filename[i], width, height, n, true);
13
14       // ---------------------------------------
15       // No Mip-Mapping - Copy Image into texture
16       glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGBA, width, height,
17                    0, GL_RGBA, GL_UNSIGNED_BYTE, image);
18       // ---------------------------------------
19
20       // Delete image data
21       delete[] image;
22    }
23    // Unbind texture
24    glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
25    // return texture
26    return texture;
27 }
```

Skybox

OpenGL cube maps can be used to easily draw an environment map (skybox).

The skybox is rendered as a cube around the viewer with a cube map used to map the textures onto the faces of the cube.

The positions of the vertexes can be used to generate the STR vectors necessary to access the cubemap.

Skybox - Vertex Shader

The vertex shader generates STR vectors from the positions of the cube's vertexes.

```
1  // OpenGL 4.0
2  #version 400
3
4  // Input to Vertex Shader
5  in vec4 vert_Position;
6
7  // Transform Matrices
8  uniform mat4 u_View;
9  uniform mat4 u_Projection;
10
11 // Output to Fragment Shader
12 out vec3 frag_str;
13
14 void main() {
15    // -------------------- Output to Fragment Shader --------------------
16    frag_str = normalize(vert_Position.xyz * vec3(-1.0f, -1.0f, 1.0f));
17
18    // -------------------- Vertex Position --------------------
19    gl_Position = u_Projection * u_View * vert_Position;
20 }
```

Skybox - Fragment Shader

The fragment shader simply needs to access the cube map (note the type of sampler) to draw the skybox.

```
1  // OpenGL 4.0
2  #version 400
3
4  // Input from Vertex Shader
5  in vec3 frag_str;
6
7  // Texture
8  uniform samplerCube u_texture_Map;
9
10 // Output from Fragment Shader
11 out vec4 pixel_Colour;
12
13 void main () {
14    // -------------------- Fragment Colour --------------------
15    pixel_Colour = texture(u_texture_Map, frag_str);
16 }
```

Skybox - View Matrix

Rendering the skybox requires a different view matrix to the other shader programs. The skybox represents very distant objects and the movement of the viewer should not affect their appearance.

Instead of a normal view matrix, the skybox needs an orientation matrix (just the rotational part of the view matrix).

This can be easily generated by simply setting the `viewPosition` to `(0,0,0)`.

Skybox - View Matrix

Functions to generate a general view matrix and the associated orientation matrix:

```
1 // Generate Position, Forward and Up Vectors
2 ...
3
4 // Generate View Matrix (for normal objects)
5 glm::mat4 viewMatrix = glm::lookAt(mPosition, mPosition + forward, up);
6
7 // Generate Orientation Matrix (for skybox)
8 glm::mat4 orientationMatrix = glm::lookAt(glm::vec3(0,0,0), forward, up);
```

The orientation matrix is fixed to stay at position (0,0,0) which is the centre of the skybox cube.

Skybox - View Matrix

As the viewer moves through the scene, their position will change in relation to the objects in that scene.

However, the view of the skybox will only change based on the direction of view and not the position of the viewer.

The skybox should only show objects for which this is true.

Skybox - Example

Example 11 - Skybox