Implementing Reflections Using Local Cubemaps

Roberto Lopez
Senior Engineer
Content

• Environment Mapping and Cubemaps
• Reflections with Infinite Cubemaps
• Local Correction
• Infinite and Local Cubemaps
• Implementation of Local Correction in the Shader
• Advantages of using Local Cubemaps for reflections
  • Offline cubemap filtering
• Limitations of using Local Cubemaps for reflections
• Overcoming limitations of Local Cubemaps
  • Runtime rendering reflections
  • Reflection camera. Reflection shader
• Conclusions
Environment Mapping

Environment mapping simulates reflections or lighting upon objects without going through expensive ray-tracing or lighting calculations.

Spherical Environment mapping

A clever way of doing reflections in real time.

Limitations:
Distortions when mapping a picture onto a sphere.

The spherical surface is mapped into 2D.
Why Cubemaps?

In 1999, it became possible to use cubemaps with hardware acceleration.

**Cubemaps**

- Hardware accelerated.
- They solved the problems of image distortions, viewpoint dependency and computational inefficiency.

Source images are sampled directly. No distortion introduced by resampling into an intermediate environment map.
Reflections with Infinite Cubemaps

Normal N and view vector D are passed to fragment shader from the vertex shader.

In the fragment shader the texture colour is fetched from the cubemap using the reflected vector R:

```plaintext
float3 R = reflect(D, N);
float4 col = texCUBE(Cubemap, R);
```
Wrong Reflections

Reflections generated using a cubemap without any local binding.
Instead of fetching the texel from the cubemap using the reflected vector \( R \) the intersection point \( P \) with the bounding sphere is found and a new vector \( R' \) from the cubemap position \( C \) to the intersection point \( P \) is built and this new vector is used to fetch the texture colour from the cubemap.

\[
\begin{align*}
\text{float3 } R &= \text{reflect}(D, N); \\
\text{Find intersection point } P \\
\text{Find vector } R' &= CP \\
\text{Float4 } \text{col} &= \text{texCUBE}(\text{Cubemap, } R');
\end{align*}
\]

Reference:
Local Correction - Better Approach

Instead of fetching the texel from the cubemap using the reflected vector \( R \) the intersection point \( P \) with the bounding box is found and a new vector \( R' \) from the cubemap position \( C \) to the intersection point \( P \) is built and this new vector is used to fetch the texture colour from the cubemap.

\[
\text{float3 } R = \text{reflect}(D, N); \\
\text{Find intersection point } P \\
\text{Find vector } R' = CP \\
\text{Float4 col = texCUBE(Cubemap, R');}
\]


Reflections generated after applying the “local correction”.

Right Reflections
Infinite and Local Cubemaps

**Infinite Cubemaps**
- They are used to represent the lighting from a distant environment.
- Cubemap position is not relevant.
- They are used mainly for outdoor lighting. For example, skyboxes.

**Local Cubemaps**
- They are used to represent the lighting from a finite local environment.
- Cubemap position is relevant.
- The lighting from these cubemaps is right only at the location where the cubemap was created.
- *Local correction* must be applied to adapt the intrinsic infinite nature of cubemaps to local environment.
Vertex Shader

vertexOutput vert(vertexInput input)
{
    vertexOutput output;

    output.tex = input.texcoord;
    // Transform vertex coordinates from local to world.
    float4 vertexWorld = mul(_Object2World, input.vertex);

    // Transform normal to world coordinates.
    float4 normalWorld = mul(float4(input.normal, 0.0), _World2Object);

    // Final vertex output position.
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);

    // ----------- Local correction -----------
    output.vertexInWorld = vertexWorld.xyz;
    output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
    output.normalInWorld = normalWorld.xyz;

    return output;
}
float4 frag(vertexOutput input) : COLOR
{
  float4 reflColor = float4(1, 1, 0, 0);

  // Find reflected vector in WS.
  float3 viewDirWS = normalize(input.viewDirInWorld);
  float3 normalWS = normalize(input.normalInWorld);
  float3 reflDirWS = reflect(viewDirWS, normalWS);

  // Working in World Coordinate System.
  float3 localPosWS = input.vertexInWorld;
  float3 intersectMaxPointPlanes = (BBoxMax - localPosWS) / reflDirWS;
  float3 intersectMinPointPlanes = (BBoxMin - localPosWS) / reflDirWS;
  // Looking only for intersections in the forward direction of the ray.
  float3 largestRayParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);
  // Smallest value of the ray parameters gives us the intersection.
  float distToIntersect = min(min(largestRayParams.x, largestRayParams.y), largestRayParams.z);
  // Find the position of the intersection point.
  float3 intersectPositionWS = localPosWS + reflDirWS * distToIntersect;
  // Get local corrected reflection vector.
  reflDirWS = intersectPositionWS - _EnviCubeMapPos;

  // Lookup the environment reflection texture with the right vector.
  reflColor = texCUBE(_Cube, reflDirWS);
  // Lookup the texture color.
  float4 texColor = tex2D(_MainTex, float2(input.tex));

  return _AmbientColor + texColor * _ReflAmount * reflColor;
}
Filtering Cubemaps to Achieve Visual Effects

The cost of the process depends on filter complexity and cubemap resolution.

Just use the filtered cubemap.
Filtering Cubemaps to Achieve Visual Effects

Reflections Based on Filtered Local Cubemap

Reflections showing a “frosty” effect.
Workflow for Offline Cubemap Filtering

1. Generate the cubemap and save it as a dds, cube cross or individual images.
2. Import the cubemap in CubeMapGen.
3. Apply filtering in CubeMapGen and save the result in a convenient format.
4. Import back the filtered cubemap into your development framework.
5. Use the filtered cubemap as any other cubemap in your game.


Detailed explanation and source code in ARM Guide to Unity.
Reflections based on Local Cubemaps: Pros and Cons

Advantages

1. Simple to implement.
2. Very realistic.
3. Physically correct.
5. Cubemap texture can be compressed.
6. Offline filtering effects can be applied which could be very expensive at run time.

Disadvantages

1. Works fine only in open plan space with no geometry in the centre from where the cubemap will more likely be generated.
2. Objects in the scene must be close to the proxy geometry for good results.
3. Valid only to simulate reflections of static geometry.
Handling Reflections of Dynamic Objects

- **Reflection from Static Geometry**
  - Use Local Cubemap technique

- **Reflection from Dynamic Geometry**
  - Use Virtual Reflection Camera technique

**All Reflections**
- Combine both types of reflections
Dynamic Objects Runtime Reflections with a Virtual Camera

Setting the camera upside down affects the winding of the geometry.

We need to reverse geometry when rendering with the reflection camera to fix the winding order.

Virtual reflection camera renders objects upside down (flipped Y)

Reflective surface

Main camera

Original geometry

Mirrored geometry
Mirror Reflection Matrix

\[ R = \begin{bmatrix}
1 - 2n_x^2 & -2n_xn_y & -2n_xn_z & -2n_xn_w \\
-2n_xn_y & 1 - 2n_y^2 & -2n_yn_z & -2n_yn_w \\
-2n_xn_z & -2n_yn_z & 1 - 2n_z^2 & -2n_zn_w \\
0 & 0 & 0 & 1
\end{bmatrix} \]

\[ n_x = planeNormal.x \]
\[ n_y = planeNormal.y \]
\[ n_z = planeNormal.z \]
\[ n_w = -\text{dot}(planeNormal, planePos) \]
Calculate reflection matrix $\text{ReflMat}$ relative to reflection plane

Calculate the position of the reflection camera:
\[
\text{reflCam.Pos} = \text{mainCam.Pos} \times \text{ReflMat};
\]

Build world to camera matrix for reflection camera:
\[
\text{reflCam.WorldToCam} = \text{mainCam.WorldToCam} \times \text{ReflMat};
\]

Set projection matrix for reflection camera:
\[
\text{reflCam.ProjMat} = \text{mainCam.ProjMat};
\]

Set render texture:
\[
\text{reflCam.SetRenderTex(\text{reflTex})}
\]
\[
\text{reflMat.SetTex(\_\text{ReflTex}, \text{reflTex});}
\]

Render reflections:
\[
\text{GL.ReverseBackFacing(true);}\]
\[
\text{reflCam.Render();}
\]
\[
\text{GL.ReverseBackFacing(false);}\]
Rendering Static and Dynamic Reflections – Vertex Shader

```cpp
vertexOutput vert(vertexInput input) {
    vertexOutput output;

    // Transform vertex coordinates from local to world
    float4 vertexWorld = mul(_Object2World, input.vertex);
    // Transform Normal to world coordinates
    float4 normalWorld = mul(float4(input.normal, 0.0), _World2Object);
    // Final vertex output position
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);

    // ----------------- Local correction -------------------
    output.vertexInWorld = vertexWorld.xyz;
    output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
    output.normalInWorld = normalWorld.xyz;

    // ----------------- Runtime Reflection texture -----------
    output.vertexInScreenCoords = ComputeScreenPos(output.pos);

    return output;
}
```
float4 frag(vertexOutput input) : COLOR
{
    // --------------- Find reflected vector in World Space ---------------
    float3 DirectionWS = normalize(input.viewDirInWorld); // Interpolated
    float3 normalWS = normalize(input.normalInWorld);
    float3 ReflDirectionWS = reflect(DirectionWS, normalWS);
    // --------------- Apply local correction to reflection vector ------
    float3 newReflDirectionWS = LocalCorrect(ReflDirectionWS, _BBoxMin, _BBoxMax,
                                            input.vertexInWorld, _EnviCubeMapPos);
    // -------------- Lookup the color in the static cubemap -----------
    float4 staticReflColor = texCUBE(_Cube, newReflDirectionWS);
    // -------------- Lookup the color in the projected texture produced by the Virtual Reflection Camera ----
    float4 dynReflColor = tex2Dproj(_ReflectionTex, UNITY_PROJ_COORD(input.vertexInScreenCoords));
    // ------------ Revert blending of reflection camera ----------------
    if(dynReflColor.a > 0)
        dynReflColor.rgb /= dynReflColor.a;
    // -------------- Combine static and dynamic reflections -----------
    float4 reflCombiColor;
    reflCombiColor.rgb = lerp( staticReflColor.rgb, dynReflColor.rgb, dynReflColor.a );
    reflCombiColor.a = 1.0;
    return reflCombiColor;
}
Combined Reflections

- Reflections from static geometry using local cubemap
- Reflections from the sky using infinite cubemap
- Reflections on the chess piece using local cubemap
- Reflections from dynamic geometry using a virtual camera
Wrap up

- We have seen the concept of local cubemaps and how they differ from the standard “infinite cubemaps”.

- We have seen how to implement reflections based on static local cubemaps: an effective technique to implement realistic and high quality reflections with a low impact on resources which is important in mobile platforms where resources must be carefully balanced.

- We have seen the advantages and limitations of reflections based on local cubemaps.

- We have seen how to combine reflection from static geometry based on local cubemaps with reflections of dynamic geometry rendered to texture at runtime.

ARM Guide to Unity available at MaliDeveloper.ARM.com
Thanks