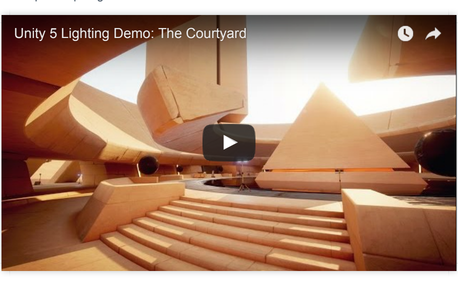
AWESOME REALTIME GI ON DESKTOPS AND CONSOLES

JESPER MORTENSEN, NOVEMBER 5, 2015

We've teamed up with Alex Lovett again and built *The Courtyard*, a demo that puts the Precomputed Realtime GI features in Unity 5 to good use. He previously built the Shrine Arch-viz demo. This time, however, the goal was to build a demo aimed at game developers requiring realtime frame rates. Check out this video:



Alex built the demo in about 8 weeks using Unity 5.2 off the shelf without any additional packages from the Asset Store – everything was built from scratch.

There is no baked lighting in this scene

The demo relies on Precomputed Realtime GI and realtime lighting throughout. It has a full *time-of-day cycle*, *emissive geometry,* about *100 animated spotlights* that come alive at night, as well as a number of *floodlights* on stands and an optional *player flashlight*. The time-of-day cycle uses an animated skybox that is synchronized with the sun in order to capture the subtle lighting changes. In the playable demo we are now making available to you (see below), a UI has been added that allows you to control all of these lighting features in-game. Here are a few shots from the scene at different times of day:

The scene was created to be especially demanding in terms of lighting. A significant part of it is lit only by bounced light, and when the sun has set it is lit almost entirely by bounced light.

The realtime GI system works by precomputing all of the *possible* paths that light can take between static objects in the scene. This makes it possible to tweak the lighting in realtime, without interruption, because the system already has all the information it needs to quickly calculate the consequences of lighting changes. However, this means that static objects should not be modified, because doing so would require precomputing all the paths again. Given this, it makes sense to author levels in stages: *geometry* then *lighting* (and then repeat if necessary). Haphazardly moving static geometry around and adjusting lighting at the same time will require many lighting builds. We are working on more progressive and interactive lighting workflows for Unity 5.x. More details on this will follow in a separate blog post.

Authoring for Realtime GI

The demo was built with desktop PCs and consoles in mind, see the blogpost on GI in Unity 5 covering the Transporter demo for Realtime GI use on mobile platforms.

The Realtime GI system in Unity 5 is powered by Geomerics Enlighten and is designed for use in games. All the lighting computations are performed asynchronously on CPU worker threads; because games are usually GPU bound, the extra CPU work has very little impact on overall framerate. Also, only the areas where the lighting has changed are recomputed.

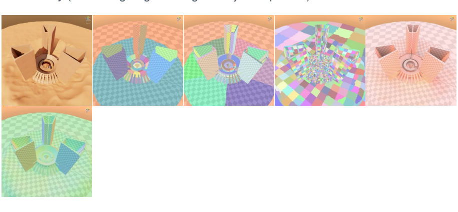
The lighting latency in the game is dependent on the resolution chosen for the realtime indirect lightmaps. In this demo Alex set the resolution to be relatively low – in order to be responsive – but such that it still captures the desired lighting fidelity and subtleties in the indirectly lit areas.

The indirect lightmap resolution was:

- One texel every two units (i.e. 0.5 texels per unit) in the central areas.
- One texel every 10 units in dunes close to the central area.
- One texel every 32 units in dunes in the outer areas.

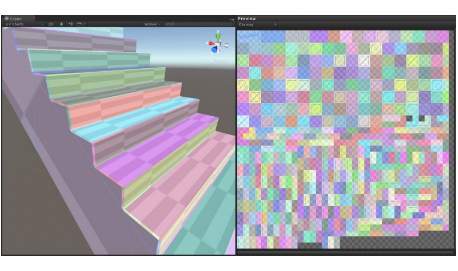
In order to balance the resolutions, an overall baseline of 0.25 texels per unit was set on the scene. Then, multipliers were added using custom lightmap parameters to give some really nice lighting and a precompute time of just 15 minutes.

The following screenshots show a *shaded overview* of the scene, the *Enlighten systems* generated, the *UV charting* view (showing the resolution of the indirect lightmaps), the *clusters* (responsible for emitting bounce lighting), the *bounced lighting*, and the *lighting directionality* (used for lighting off axis geometry and specular):



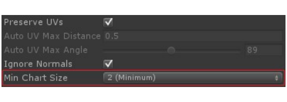
Care was taken to provide good lightmapping UVs. In some cases they were carefully authored to make sure that the models perform well for both lighting builds and the runtime. One specific instance of this is the staircase model.

Staircases can be difficult to get right, since at large texel sizes a texel can cover more that a single step. This can cause lighting levels to vary unexpectedly between the steps. On the other hand, using many texels for the steps becomes expensive in terms of performance. The staircase used in this scene also had bevels, which can really throw off the unwrapping and packing for realtime GI and generate many unnecessary charts taking up texel space. The initial staircase design looked like this in realtime GI UV layout:

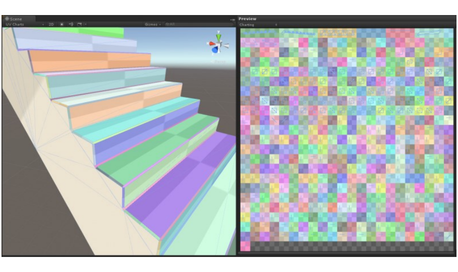


This takes up a 70×72 texel realtime lightmap. There are two problems with this layout. Firstly, it uses too many texels per step (4×4); secondly the bevels are split into separate charts that also take up a minimum of 4×4 texels. Why can each chart not just use 1 texel? Firstly, Enlighten is optimized to use 2×2 texel blocks when processing the textures in the runtime, so every chart must be at least 2×2 texels. Secondly, Enlighten includes a stitching feature where charts can be stitched together to allow smooth results on, for example, spheres and cylinders; this feature requires that a chart have separate directionality information at each edge. Directionality information is only stored on a per-block basis, so a stitchable chart will always need a minimum of 2×2 *blocks* – becoming a minimum of 4×4 texels. Since no stitching is needed for the staircase, 2×2 texel charts suffice.

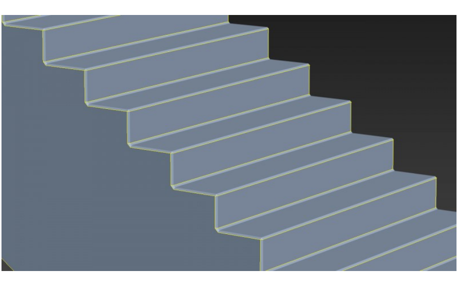
We have introduced an option for this in the Object properties of the lighting panel:



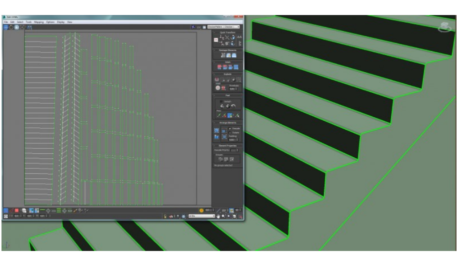
The value can either be 4, which works well for stitching in a setting that uses directionality, or 2 which is more compact. Setting the minimum chart size option to 2 reduces the texel density significantly – now the model fits in a 44×46 texel realtime lightmap:



The bevels are still taking up unnecessary chart space. This is somewhat unexpected as bevels and steps have been authored such that the bevel is part of the step in UV space. The image below shows the UV borders overlaid on the model. Notice how the bevels are integrated into the steps:



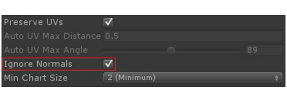
In the 2D view of the lightmapping UVs the bevels do not show up. This is because they have been collapsed into the step charts so they do not have any area in UV space. This is done to avoid that the lighting simulation takes the sloped bevels into account.



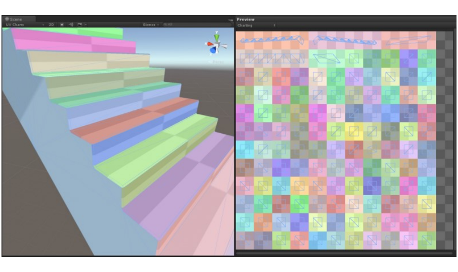
The reason that the bevels are split into separate charts is that the UVs used for realtime GI have to be repacked at the actual resolution each instance will be used at. The packing algorithm guarantees that there is a half texel boundary around charts, such that bleeding artifacts between charts are avoided. This ensures very good use of the atlases and no need for packing margins at the expense of having per instance UVs.

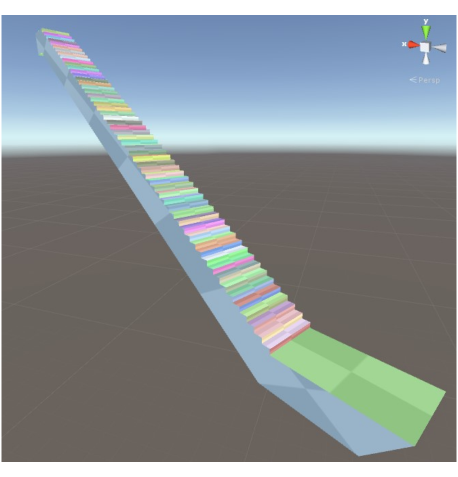
The problem arises when there is a hard edge, like in this example where there is a hard edge between the steps and the bevels. The model importer will already have duplicated the vertices at the hard edge since each side of the hard edge needs different normals. So the charts have already been split at the outset. Repacking this will cause the bevel to end up in a separate chart. This is because chart detection uses normals for detecting charts by default. When it finds an edge with identical vertex positions, identical UVs, but different normals, it will split the chart along that edge.

In this instance that behavior is not desirable. The bevel should be integrated in the chart for the step, since it will not have a significant impact on the lighting. In order to do so, we have exposed an option to ignore normals when doing chart detection during the packing step. This will keep the charts together as authored in the UVs regardless of the hard edge:



Enabling the packer option reduces the texel density even further. Now the bevels are integrated with the steps. The final result fits in a 22×24 lightmap:





By using these new options where appropriate the realtime GI precompute time could be reduced from an hour and a half to just 15 minutes.

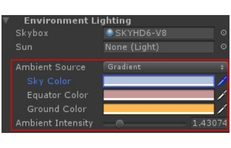
What about performance?

The Realtime GI did not need much tweaking to fit within the desired runtime performance/memory budget. However, the post-effects stack is quite deep and includes *Filmic Vignette*, *Bloom*, *Tonemapping*, *Lens Distort*, *Screen Space Ambient Occlusion*, *Color Correction Curves*, *Noise And Grain*, *Color Grading Properties* and *Antialiasing*. Despite this the level runs at 60fps on a fast desktop with a decent GPU.

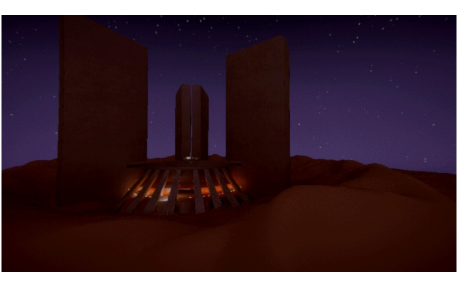
There were however a few tweaks applied to realtime GI.

Fast Environment Lighting

The realtime GI system is capable of using a skybox directly to drive the environment lighting inputs. However, using this feature requires downloading the skybox textures from the GPU in order to update the CPU based realtime GI system. This is not ideal when the environment lighting changes every frame as it does in this demo when the time-of-day simulation is running. Instead, the environment lighting is derived from the current time and converted to a lighting gradient and ambient intensity that is used to drive the realtime GI system. These are the settings in the Lighting panel that are used to drive the gradient based ambient inputs:



A gradient ambient source can be handled entirely on the CPU. This is very performant and gives a result that is nearly indistinguishable from using the full skybox.



The bare bones code used to update the environment lighting looks like this:

```
using System;
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[ExecuteInEditMode]
public class EnvironmentUpdater : MonoBehaviour {

  public Gradient groundGradient, equatorGradient, skyGradi
  private TimeOfDayManager m_TimeOfDayManager;

  void OnEnable () {
    m_TimeOfDayManager = FindObjectOfType<timeofdaymanager>
  }

  void Update () {
    float currentTime = m_TimeOfDayManager.time;
    RenderSettings.ambientGroundColor = groundGradient.Eval
    RenderSettings.ambientEquatorColor = equatorGradient.Ev
    RenderSettings.ambientSkyColor = skyGradient.Evaluate(c
  }
}
```

The code assumes a manager that handles time-of-day. This demo has such a manager that is hooked up to the UI so you can adjust time-of-day interactively.

Fast Emissive Lighting

In a vein similar to the environment lighting the realtime GI system is capable of driving emissive objects directly from the emissive shader properties. It will by default render out an emissive map in realtime lightmap space using the emission material property or, alternatively, a customized shader meta pass can be added. This will require a texture download from the GPU before the data can be consumed by the realtime GI system. In order to avoid this, there are script bindings available to set the emissive property directly on the object as a fixed color (DynamicGI.SetEmissive) and allow the realtime GI system to apply the emission directly on the GI simulation. This bypasses the GPU completely and provides a way to light the scene nearly for free. Here is a few mood shots using this feature:

How does it scale with larger worlds?

Clearly this is not a massive world – so how does it scale? In order to keep the memory footprint low and the lighting responsiveness high it makes sense to dice up larger worlds so that parts of them can be streamed in and out while the player navigates the world. The Realtime GI system works with LoadLevelAdditive and UnloadLevel. Of course some care needs to be taken when unloading levels since levels that are not directly visible may still affect the bounced lighting significantly. We are looking into adding some scripting hooks for enabling bounce fade between levels before unloading, so that lighting pops can be avoided, thereby allowing for more aggressive streaming. Another thing that we are looking into is providing scripting control for prioritizing update frequency for instances *within* the level for more fine-grained control beyond level streaming.

Using additive loading, or the Multi-Scene Editing feature currently scheduled for release in Unity 5.3 on December 8, will allow you to easily build scalable worlds lit by beautiful Realtime GI.

How do I get this demo?

You can download a prebuilt player here for OSX and Win64 and finally for Linux. In addition you can download the Project from the Asset Store. The project works with Unity 5.2.2p2 and later. The controls are described in a text file in the player zip file.

The demo runs best on a desktop machine. The timings listed here were achieved with an Intel i7-4790 3.6GHz machine with 16GB RAM, NVIDIA GeForce GTX 780 GPU, running on Windows 10.

Credits

Alex Lovett (aka @heliosdoublesix) creative-directed, art-directed, built, lit, animated, audio-directed and delivered this demo while tirelessly stress testing GI features in Unity.
Thomas Pasieka and **Plamen 'Paco' Tamnev** modelled the props.
Music Marks composed the music for the video preview.
Dave Dexter composed the audio for the playable.
Silvia Rasheva supported the demo as a producer.
Morgan McGuire fixed some reflection issues.

Jesper Mortensen, **Kuba Cupisz** and **Kasper Storm Engelstoft** supported Alex during development, and used the collaboration to further improve the lighting system in Unity for all artists out there.

Special shout-out to **Plamen 'Paco' Tamnev** for fixing some last minute issues for us.