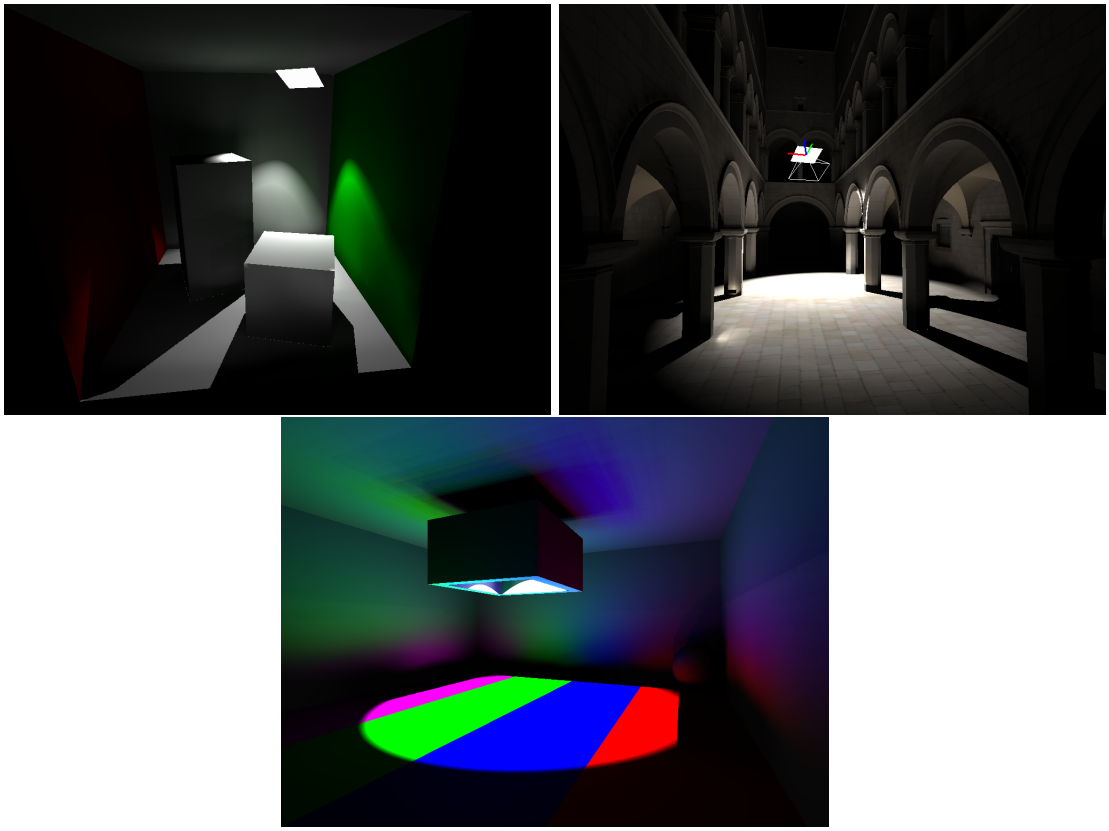
T-111.5310 Vuorovaikutteisen tietokonegrafiikan jatkokurssi

Assignment 3: Instant Radiosity

Due 28.4. at 23:59.

In this assignment, you will implement a basic instant radiosity algorithm. The algorithm runs in (near) real time, but on the other hand it is biased and generally does not give as high quality results as e.g. the radiosity solver in the last assignment. The idea is to shoot a few dozen rays from the light source, and to place virtual point lights at the hit points. These point lights give an approximation to the first bounce of indirect illumination. All lights must be shadowed, so you will also implement shadow mapping using OpenGL shaders.

The program uses a single spot light as the primary light source. The instant radiosity solver adds the indirect illumination effects. The result looks like this:



The requirements are:

1. Compute the shading caused by the spot light source (1p).
2. Implement basic shadow maps (4p).
3. Implement the instant radiosity algorithm (5p).

1 Getting Started

1.1 Sample solution

As always, we provide a sample solution executable (`solution.exe`). See how it works, and try out the effects of the various sliders and buttons. Notice in particular the following features: light intensity and FOV sliders, tone mapping sliders, light source position visualization toggle, render-from-light-toggle, shadow map resolution slider, and the slider adjusting the number of light sources cast by the instant radiosity solver.

This time we provide three scenes, which can be loaded by number keys 1, 2 and 3. The first two scenes are the same as in the last assignment. The third scene is the nice texture color test scene posted on Piazza (thanks Daniel!)

1.2 Starter code

We provide quite a bit of starter code for this assignment. Much of the general infrastructure for multipass rendering, shader compilation, indirect light source lists, etc. is in place. The reason for this is that managing OpenGL resources is a bit involved, and we wanted to keep the focus of the assignment on the algorithms, not on technical API programming details. However, we encourage you to look through the code to see how the whole thing works. In particular, read the function `App::renderFrame()` carefully, and follow the function calls it makes to see what steps the full algorithm takes, and where your input is required. You should not need to concern yourself with the OpenGL technicalities (anything involving allocation and freeing of resources), unless you specifically want to change how it works.

The sections requiring code by you are marked with the comment "your code here". Of course, you're free to add and modify code in other places too. There are a quite a few comments with further instructions and explanations in the code.

The assignment makes heavy use of OpenGL shaders. The shaders are defined as inlined GLSL language code inside some of the functions. The shader skeletons have a lot of comments that explain what is happening.

You will need to integrate your ray tracer from the first assignment again. Alternatively, you can use the ready-made raytracer library, but this will again cost you three points. This time the ray tracing library is included in the same package as the standard starter code. To use it, choose the `Release_RTLIB` or `Debug_RTLIB` configuration from the Visual Studio configuration menu.

2 Spotlight shading (1p)

The starter code does not use the light source for anything. You'll need to implement the light shading computations to the main fragment shader. The shader source code is inlined within the function `LightSource::renderShadowedScene()`. The function `renderShadowedScene()` implements the main rendering pass, where the entire scene is drawn from the camera's viewpoint. Have a look at `App::renderFrame` and see how it calls this function.

You'll need to add the shading computation into the fragment shader. The starter code already imports all the required variables (such as light source position and direction, in eye coordinates) in uniforms, as described in the code comments. The shading is a product of the following terms:

- Light emission.
- Surface color.
- $1/\pi$, from the diffuse BRDF.
- Inverse square of the distance from the surface point to the light. (As a practical kludge, you may want to limit the maximum value of this term to, say, 10, after you've implemented the instant radiosity

solver. You'll see that the indirect lights can cause some very bright spots if they land near corners; it is due to this term.)

- Cosine of the angle between the surface normal and the incident lighting direction.
- Cosine of the angle between the light normal and the incident lighting direction. This is because we assume that the light is a diffuse emitter.
- Spotlight emission distribution, i.e. the circular cone.

The value of the last term should be 1 if the point is inside the cone of the light source (except maybe at the edges), and 0 outside it. The opening of the cone is called the field of view, or FOV. Try to come up with a way to make the transition edge fade from 1 to 0 smoothly, so that you'll get a smooth spotlight edge.

(For the sample code, we used the function $\max(0, \min(1, 4 * (\cos\alpha - \cos(f/2))/(1 - \cos(f/2))))$ for the cone term. Here f is the FOV in radians, and α is the angle of the incident illumination versus the light cone axis. It is heuristic and messy, but it gives a decently smooth edge falloff and a correct opening angle. But try to think about it yourself instead of blindly copying the formula.)

3 Shadow mapping (4p)

Accounting for light visibility is a crucial part of realistic rendering. The most obvious visibility features are shadows. In the previous radiosity exercise, a shadow appeared whenever there was a blocker between a surface point and the light source; this was detected using the ray tracer. Indeed, ray tracing is a robust and high-quality method for rendering shadows. However, ray tracing is generally too expensive to be used in real time applications. Instead of using the ray tracer, you will implement a technique called shadow mapping. Shadow mapping is a very popular method for rendering shadows. It is much more suitable for hardware rendering than ray tracing. It is used by most games, and it was even the preferred method in off-line film rendering up to recent years.

The idea is to first run a pre-rendering pass, where the scene is rendered from the viewpoint of the light. The depth values are stored to a texture map called the shadow map, which might look like this:

In the main rendering pass, we perform a shadow test at every pixel by computing the depth from the light, and comparing it to the depth value stored in the corresponding pixel of the shadow map. If the distance is larger, then we know that there is another surface point closer to the light. Hence the light is blocked, and our point is in shadow.

3.1 OpenGL transformations

We first need to figure out how to render the scene from the light's point of view. Note that spot light sources are not really different from cameras in the geometric sense. Both have a position, orientation and a field of view. Indeed, we can use the familiar camera transformations to render from lights also.

Let us briefly recall how OpenGL viewing works. More information can be found e.g. in the slides of the basic course. The camera position, orientation and viewing angle are defined by two matrices: the view matrix $\mathbf{V} \in \mathbf{R}^{4\times4}$ that transforms points from the world coordinates to eye coordinates, and the projection matrix $\mathbf{P} \in \mathbf{R}^{4\times4}$ that transforms points from eye coordinates to the clip coordinates (accounting for the field of view and the near and far planes). When rendering through a camera or a light (let's just use the word camera for now), we perform the following steps:

- Rotate and translate the homogeneous world-coordinate points $\mathbf{p} = [x\ y\ z\ 1]^\top$ to the eye coordinate space, i.e. the space where the camera is at the origin and looks along the negative z-axis. This is achieved by multiplying the vertex coordinates by $\mathbf{V}$.
- Transform the eye space points to clip space by multiplication with $\mathbf{P}$. Let $\mathbf{c} := \mathbf{PVp}$ be the world-space point transformed to the clip space using the aforementioned matrices.
- The clip space coordinates $\mathbf{c}$ determine both the screen coordinates and the depth as follows. The vector $\mathbf{c}$ is homogeneous, i.e. a four-component vector $\mathbf{c} = [\mathbf{c}_x\ \mathbf{c}_y\ \mathbf{c}_z\ \mathbf{c}_w]^\top$. We get a usual three-dimensional point by dividing by the w-component: $\tilde{\mathbf{c}} := \mathbf{c}_{xyz}/\mathbf{c}_w$ (the keyword here is projective transformation). After this division, the point $\tilde{\mathbf{c}}$ is in so called normalized device coordinates (NDC). Due to how the projection matrix was defined, the points that were inside the camera viewing frustum will end up with NDC coordinates in the range $[-1, 1]$. So a point with NDC coordinates $[-0.9\ 0.9\ 0.3]^\top$ will be drawn somewhere near the upper left corner of the screen. Another point with the coordinates $[-0.9\ 0.9\ 0.1]^\top$ will be drawn in the same pixel, but it will overwrite the previous one because it is closer (its z-value is smaller).

So in summary, on a very high level: the OpenGL transformation pipeline grabs your camera's viewing frustum (the "clamped pyramid") by its eight corners, and stretches and squashes everyting inside it into the shape of a unit box $[-1, 1]^3$ (the NDC space). This transformation is done to all points in the scene, and from the results we can directly read the screen coordinates and the depth. For exhaustive details and some potentially useful illustrations, see `http://www.songho.ca/opengl/gl_transform.html`

In the starter code, we always pass the projection and view matrices to the shaders in a single pre-multiplied matrix $\mathbf{T} = \mathbf{PV}$. This is simply because in our case we always use them together anyway, and it is more efficient to just compute the product once and keep it. So whenever you see an uniform like `mat4 posToClip` or `mat4 posToLightClip`, it contains the product of these two matrices (or will contain, once you implement the matrix construction). So it's a transformation directly from the world to clip space.

In GLSL shaders, the main task of the vertex shader is to assign the clip space coordinates of the vertex to the `gl_Position` variable. Check out the existing GLSL shaders in the starter code to see how this happens.

3.2 Shadow mapping transformations

So now, the first two NDC coordinates determine where on the screen the pixel will be drawn. The third NDC coordinate determines how far it is from the camera (it's not a linear distance function due to projective distortion, but it is at least monotonous, so it can be used for depth comparisons). This directly suggests a way to implement shadow mapping:

- Render the scene from the light's viewpoint by using a view matrix that transforms the scene to the light source's eye space, and a projection matrix that is constructed from the light's field of view (the spotlight angle of opening) and near and far clipping distances (for our purposes, you can use some reasonable values like 0.01 and 100; note that there are member variables for these in `LightSource`). Compute the NDC coordinates of the scene points, and store their z-value in the depth map for each pixel.
- In the main rendering pass, i.e. when rendering the scene from the camera's viewpoint, we perform the exact same light transformations as we did in the pre-render pass. However, we do not use the result

to set the position of the vertex, because we want to use the camera's viewpoint this time. Instead, we again compute the NDC z-value but this time compare it with the corresponding value stored in the shadow map. If the value is larger than the one in the shadow map, then we know that there must have been a surface point that was closer to the light than our current point. One question remains: how do we know which pixel to look at in the shadow map? Recall that the NDC xy-coordinates determine where in the screen (or in the shadow map) the point is drawn. So we can simply use the light-NDC xy coordinates to determine the UV coordinates for the shadow map texture access. (Note that they must be re-scaled from the range $[-1, 1]$ to $[0, 1]$, because the convention is that UV coordinates are in the latter range.)

3.3 Implementation of the depth pre-pass

The starter code already runs the pre-rendering pass for the shadow map and handles the depth texture allocation and management. However, it does not use the correct transformations yet, and the shaders do not yet perform the requried functions. Overall, completing the shadow map support mostly requires you to fill in the missing transformations and to use them correctly. This amounts to perhaps ten lines of code scattered around `ShadowMap.cpp`; however, getting everything to work correctly might be a bit frustrating if your understanding of the OpenGL transformations is shaky. It might pay off to take some time to carefully review how they work.

The pre-rendering pass is done in the `LightSource::renderShadowMap()` function of the light sources (it is called from `App::renderFrame`). The output of the pass is a shadow map texture, which is stored by the light source class. The function contains the inline source for a GLSL shader that will perform the depth computation, once you implement it.

The depth shader takes as an input the transformation matrix `posToLightClip` from world coordinates to the light clip coordinates. As discussed above, this matrix is the composition of the projection matrix and the view matrix of the light. The matrix is formed in the function `LightSource::getPosToLightClip()`, which you will need to implement. The light position is encoded by its member variable `m_xform`, which is a matrix that transforms the light source from the origin into its world position. Our goal is the opposite (or one might say, suggestively, *inverse*): we wish to transform the world into the light's eye coordinates. Your task is to figure out the light view transformation based on `m_xform`. The projection matrix can be constructed using `Mat4f::projection()` with suitable `LightSource` member variables as parameters. See the previous sections for the explanation on the transformations.

The UI contains a "Render from light source view" button, which renders the scene using the matrix returned by `getPosToLightClip` of the main light source. It might turn out handy in debugging.

After this you need to make the shader output the depth values. Notice how the starter code computes the clip coordinates of the vertex and assigns them to `gl_Position` (this part is done in almost all shaders). Recall that you wanted to store the NDC-coordinate z-value of the vertex in the shadow map; computing this value requires just one more step. Figure out how to compute it, and assign it to `depthVarying`. The existing fragment shader will put it into the shadow map.

A debugging hint: Notice the "Shadow map visualization intensity" slider on the right. When you set it to some value, you'll see a small box on the lower left corner. It displays the current shadow map contents for the main spotlight, multiplied by the slider value. However, unfortunately the map might look almost completely gray when the shadow map is correct. This is because we store projectively distorted depth values instead of the "linear depth". But it might be useful in debugging. Debugging the shadow maps (and shaders in general) is a bit challenging because it is difficult to output debug information. You'll need to be creative, e.g. you may try coloring the surfaces based on some informative values and seeing what happens.

3.4 Implementation of the shadowed main render pass

We now have a depth map texture that has been rendered from the light's point of view. Next it is time to implement the main render pass from the camera, with shadows included. The relevant part is the `LightSource::renderShadowedScene()` function, and the already familiar GLSL shader defined inside it.

Locate the line where `App::renderFrame()` calls it, so you'll know what the big picture is. Notice that this shader too receives the `posToLightClip` uniform, which contains the same transformation you used in the shadow map render pass. This time it will not be used for positioning the camera; instead, it is used to compute the depth-from-light values that will be tested against those stored in the shadow map.

The depth computation is again implemented in the vertex shader. Use the `posToLightClip` matrix and the vertex position `pos` to repeat the same depth calculations the depth pass shader did (but of course, do not use it to set the vertex positions.) Place the depth value into `shadowDepth`. Read the earlier section on the transformations, and figure out what the UV coordinates for the shadow map access should be. Assign them to `shadowUV`.

That's it for the vertex shader. The actual shadow test is then performed in the fragment shader. The shadow map texture is readily attached to `shadowSampler` uniform, so you'll just need to fetch its value at the point `shadowUV` using the GLSL function `texture2D`. If the value stored in the shadow map is smaller than the value in `shadowDepth`, then we know that there was something else between this point and the light source. In this case, color the pixel black.

3.5 A few words on shadow maps

As can be seen even in the example solution, shadow maps in their basic form do not always look pretty. The finite resolution of the maps often shows as sawtooth edges, in particular with large FOVs. It is also difficult to get smooth edges (blurring the shadow map itself doesn't help; can you see why?) A major difficulty is so called shadow acne, where a non-shadowed point is erroneously shadowed because the depth value in the shadow map was sampled at a slightly different surface point, with a slightly different depth. One way to combat this is biasing the depth, e.g. by shadowing the point only if the depth difference is more than 1% or so (less naive methods exist too). The sample solution does not do much in this regard. However, we are using one trick, which is also activated by default in the starter code: when rendering the shadow maps, we cull the *front* faces of the surfaces, so that it is the backfaces that cast the shadows. This moves the problem to the dark side of the objects, but has its shortcomings too. Also, the front-culling might be confusing during the debugging, so you might want to turn it off initially.

In general, there is a lot more to be done to improve the shadow map quality. See the extra assignments for more information.

4 Instant radiosity (5p)

Now that we can render lights with proper shading and shadows, let us turn to the instant radiosity algorithm. The idea is to shoot rays from the primary light source, and to attach "bounce" light sources to the surface points they hit.

4.1 Overview of the starter solution

In order to spare you from the unpleasant details of OpenGL resource management, we provide the overall structure of the solver in the starter code. The state of the instant radiosity is contained in a class called `InstantRadiosity`. The `App` class owns a single instance of it, `m_instantRadiosity`.

Again, familiarize yourself with the function `App::renderFrame()`. This is the function that is called on every frame, and which coordinates all the rendering steps. The major steps it takes are as follows:

- Calls `m_lightSource->renderShadowMap()`, which renders the shadow map for the main light. This is what has been initiating your shadow rendering in the previous tasks too.
- Calls `m_instantRadiosity.castIndirect()`, which uses your ray tracer to shoot a bunch of rays from the main light source, and stores light sources on the ray hit points.
- Calls `m_instantRadiosity.renderShadowMaps()`, which simply loops through all the indirect lights from the last step, and renders a shadow map for each.

- Calls `m_lightSource->renderShadowedScene()` to render the scene using the main light source and its shadow map.
- Loops through all indirect light sources shot in the second step, and renders the scene with each light in turn. This is done with additive blending, so that the light from each source gets added on top of the previous ones. This is done because the current shader cannot handle more than one light at the time. The end result is an image with all the lights rendered.

Your task is to implement the functions `InstantRadiosity::castIndirect()`, `LightSource::sampleEmittedRays()`, and `m_instantRadiosity.renderShadowMaps()`. See the comments inside the functions.

The indirect lights use the same `LightSource` class as the main spotlight, except with a much wider FOV (150 degrees by default, but adjustable in the UI). Adding them does not require any changes to the shadow map code, if it is correctly implemented. Wide spot lights are not ideally suited for this purpose, as their resolution becomes distributed less than optimally (large pixels in the center, small pixels towards the edges). However, alternatives are much more complicated.

4.2 Implementation

The function `InstantRadiosity::castIndirect()` is called in the beginning of each frame. It takes as its parameter the number of indirect lights to be emitted. It first requests a set of samples from the main light source using the function `LightSource::sampleEmittedRays()` (more on it below). The reason that the sampling is done in bulk, instad of individual rays, is simply that this way it is easier to implement e.g. stratification, if desired. The function returns a list of ray origins, directions (scaled to maximum length, not unit length), and intensities. The intensities also have the probability density value multiplied in; more on this below. After receiving the list of rays, you must trace them into the scene, and fill the vector `m_indirectLights` accordingly. (By the way – do not use `std::vector::push_back()` on `m_indirectLights`. For technical reasons, the vector is already initialized to correct length. See the comments.)

For rays that miss the scene, you should disable the corresponding light (in fact, the starter code does this already). If a ray hits the scene, you must figure out the following things and configure the light accordingly:

- Where is the hit point?
- What direction should the light point to? Clearly, towards the hit point surface normal, but be careful about backfaces. You'll need to use the `LightSource::setOrientation()` function, which takes in a rotation matrix. For this you'll need your `formBasis()` again.
- Set the FOV based on the FOV member variable in `InstantRadiosity`.
- What is the emission? It is the entry in the emission intensity list returned by the sampler, multiplied by the surface color – so if there's a texture, you'll need to fetch the value. Otherwise use the surface diffuse color.

You can probably recycle quite a bit of code from the previous assignment. Once you activate the light by `setEnabled(true)`, it will be included in all the render passes.

The function `LightSource::sampleEmittedRays()` fills the parameter vectors with `num` entries corresponding to `num` rays, as described below. Obviously, all the rays will have the same origin, the light source position. The light should only emit rays directed within the circular cone defined by its opening angle (FOV) f. To generate such rays, you can use the a similar procedure as with ambient occlusion in the first assignment:

- Pick a point in the xy-plane unit circle e.g. by using rejection sampling.
- Scale the point coordinates by a factor r that depends on the FOV, so that your samples will lie on a smaller circle of radius r. We'll return to the choice of r shortly.

- Lift the points up to the unit sphere by setting $z = \sqrt{1 - x^2 - y^2}$. Think about what happens here: because the original points were on a circle of radius r, the lifted points are unit vectors that lie within a cone with the opening angle $\alpha = 2 \sin^{-1} r$ (you may want to draw a picture to see where this comes from). Now, we'll let you figure out by yourself how to choose r in the previous step, so that you'll get a cone with the desired angle f. It's basic trigonometry.
- Set the ray emission value to the light source's emission, divided by the number of light sources we are shooting (because the light should be distributed evenly among them).
- Here's a bit more subtle part. We emit a given number of lights regardless of the cone opening angle. However, clearly we are emitting a smaller amount of light in total if we use a narrow cone. To compensate for this, you can multiply[1] the emission value by r^2. This compensation can be derived by considering the fact that we're picking the values from a circle of area πr^2 instead of the full unit circle of area π.
- Multiply the ray lengths by the maximum length (you can use the far plane distance of the light).
- Rotate the rays towards the light normal. Again, `formBasis()`.

Note that the unit sphere lifting again induces a cosine distribution on the samples. This corresponds to our choice of including the cosine term in the shading too (recall the first part of the assignment).

`m_instantRadiosity.renderShadowMaps()` should be a simple loop that calls the shadow map rendering routine for every enabled light source.

The UI contains a button "Visualize indirect light sources" that draws the local coordinate frame at every indirect light position. Use it for debugging. Note that, confusingly, the z-axis (blue) will point to the opposite direction from the light emission, because in OpenGL we look towards the negative z-axis.

5 Extra credit

UPDATED (18.4.):

- The current implementation is quite slow because it renders the same scene dozens of times per frame. There are two ways you can make this more efficient.

  1. (2p) Modify it so that the main shader handles multiple lights in a single draw, or perhaps even all of them.
  2. (3p) Think about what the shader is actually doing: you reconstruct the 3D point that corresponds to the surface under a particular pixel, transform it to the light's projection plane, and perform a shadow test. Crucially, for a particular pixel, this point stays the same for each light, but you are still transforming and rasterizing all the triangles every time. Clearly this is a waste.

     You can get around this by precomputing the point, its normal and albedo for each pixel and storing them in off-screen floating point "G buffers" (G stands for geometry). When rendering the light, you merely look the point, normal, and albedo up from the G-buffer so you don't have to rerender the actual scene geometry again. Google "deferred shading" to find out more.

- (1p) Use a stratified or QMC pattern in emitting the indirect light sources.
- (2p) Implement Percentage Closer Filtering for both the primary light and VPLs

  See `http://dl.acm.org/citation.cfm?id=37435` and `http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html`

[1] UPDATED (15.4.): The original assignment accidentially said "divide" here. This is wrong, because clearly $r \leq 1$ and we want *less* light, not more, when the cone is narrow.

- (3p) Implement fake soft shadows for the primary light source through some suitable technique, e.g. `http://developer.download.nvidia.com/shaderlibrary/docs/shadow_PCSS.pdf` . You can find lots of other alternatives in the French survey:`http://maverick.inria.fr/Publications/2003/HLHS03a/index.php`

- (3-7p) As mentioned, the planar perspective projection distributes the resolution poorly for wide-field-of-view lights. A so-called parabolic projection works much better, and covers the entire 180 degrees (which the planar perspective cannot do at all). The problem is that straight lines become curved under the parabolic projection, so the geometry must be very finely tessellated for this to work. You can either pre-tessellate the model, which will earn you 3 points; you can also write a "geometry shader" that dynamically analyzes the incoming geometry and adaptively tessellates the geometry *inside the pipeline* as necessary. This will earn 4 more points. See`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.2283`

- (4p) Sampling all VPLs at each pixel is a lot of work. "Interleaved sampling" partitions the screen into tiles of NxN pixels (e.g. N=4 or so) and uses a *different subset* of VPLs for each pixel inside the tile. (This means that pixels $(1 + i * N, 1 + j * N)$, with $i, j >= 0$ all use the same subset, but the pixels $(1 + i * N, 2 + j * N)$ use a set different one.) This is close to Monte Carlo sampling; you think about using different random light samples for each pixel inside the tile. Obviously, this results in a lot of noise, which has to be blurred out using a bilateral filter later. See Section 5.2 of theLaine et al. paper referenced below for details; you can also look at other filtering methods, such as`http://dl.acm.org/citation.cfm?id=1921491`.

- (6p) In a static environment and with a slowly moving light source, VPLs (and their shadow maps) from the previous frame can often still contain relevant information. Implement the Incremental Instant Radiosity algorithm by Laine et al. to take advantage of this coherence. See`https://mediatech.aalto.fi/~samuli/publications/laine2007egsr_paper.pdf`and note that the 1st author's webpage has additional resources. Note that you can easily use Jonathan Shewchuk's Triangle library (`http://www.cs.cmu.edu/~quake/triangle.html`) for computing Voronoi diagrams. Note that the authors make use of interleaved sampling as well.

6 Submission Instructions

You are to write a README.txt that answers the following questions:

- Your name and student number at the beginning of the document.

- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.

- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.

- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.

- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.

- Got any comments about this assignment that you'd like to share?

You should create a single .zip archive containing:

- Your whole Visual Studio project.
- The README.txt file.

Submit it online in Optima by 28.4. at 23:59.
This assignment does not require the submission of an artifact.