Procedural Window Lighting Effects for Real-Time City Rendering

Jennifer Chandler*
University of California, Davis

Lei Yang
Bosch Research North America

Liu Ren
Bosch Research North America

Abstract

We present a new method for procedurally generating window lighting and building interior effects for real-time rendering of city scenes. Our method can selectively light a random subset of windows on the building with varying color and brightness. This window light can be combined with cube maps of representative building interior (i.e. interior maps) to generate rich details at different scales in large-scale city rendering. Our method relies solely on window transparency cutout (alpha) layer and a simple texture parameterization scheme in the building models and requires no additional information to be stored per building. The algorithm can be implemented completely in a fragment shader function, which makes it suitable for integration into existing render engines. The percentage of window lighting can be adjusted smoothly using a single parameter. We demonstrate our method using a large-scale 3D city data set, running with interactive frame rates on both PC and mobile platforms.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture;

Keywords: procedural rendering, interior mapping, window lighting, night-time rendering, facade parameterization

1 Introduction

Procedural methods have proven very successful in synthesizing both interiors and exteriors of buildings in large-scale city scenes [Smelik et al. 2014]. Such rule-based methods are capable of generating plausible geometry contents for building shapes, facades, and interior floor plans on very large scales with reasonable variance and detail. However, the synthesis of lighting effects for buildings and city scenes has been mostly overlooked. Especially in night and dark scenes, window lighting and revealed interior largely affect the appearance of city scenes at different viewing scales. Generating these lighting effects has equal importance to the final rendering of the scenes as other geometry contents do.

Existing procedural building methods and tools can be used to generate random room lighting and interior geometry to match the exterior of buildings. Such contents are usually represented by geometry primitives and are therefore static. In realistic visualization, lighting usually changes as time progresses (e.g. more room lights will turn on at dusk and switch off late at night). Therefore each window light needs an individual representation. Some of these data can be precomputed and stored in textures, such as window light masks. Unfortunately such texture contents require extra storage space and can easily result in visible repetition in rendering if the texture is shared among multiple facades.

Moreover, the majority of existing building models, such as those used in geo-visualization applications and various video game contents, do not possess interior geometry and lighting data. Such data are not trivial to generate automatically, since the semantic information associated with the geometry may not be available. Buildings with irregular floor plans and room sizes make it difficult to infer the correct layout of interior geometry. Previous methods for generating these contents (e.g. Interior Mapping) assume a uniform or predefined grid size of interior structure. While these methods can be extended with explicit information of room layout and size to support more varieties, storing and accessing such information during rendering making them less flexible to be scaled up or applied to existing pipelines.

We present a method to automatically render random window lighting and interior maps without explicit information of room sizes and locations. The key to our algorithm is a method to infer room locations based on texture coordinates of the facades. Since all the computation is done at run-time in a fragment shader, our method does not require precomputing and storing either the size or the location of rooms, and it easily scales to handling scenes of arbitrary sizes.

Our method does require that the texture parameterization of facade images follows certain rules. Specifically, each floor of the building must have a unique range of vertical texture coordinates. We started our development with 3D city data from a major GIS data provider, and the data had already been modeled based on this rule. With the `repeat` texture addressing mode, the UV layout of arbitrary existing building models can be adjusted to follow this rule without affecting the appearance. This requirement can be further generalized to 2D (i.e. both vertical and horizontal range) for more precise definition of room scales. We further require that the window areas in the texture are marked as transparent. Such layer can either be provided by texture artists or generated automatically using computer vision algorithms [Musialski et al. 2013].

To summarize, the main contributions of this paper include:

- A scheme to define interior room span in buildings using facade texture coordinates
- A method to generate pseudo-random lit window patterns on facades using window curtain functions
- A method to place room separation walls in combination with window curtain functions
- An algorithm to compute world-space room coordinates using screen-space derivatives, with interior mapping integration

Our methods can be easily integrated into existing renderers and provide convincing, adjustable results. It can potentially benefit video games, geo-visualization and any related applications by enriching the night appearance of large-scale city scenes. Additionally, our interior mapping extension could be coupled with a level of detail approach when the scene contains actual interior geometry.

*This work was mainly carried out during June–September 2014, when the first author was an intern at Bosch Research.

I3D 2015, February 27 – March 01, 2015, San Francisco, CA.

ACM 978-1-4503-3392-4/15/02 $15.00
http://dx.doi.org/10.1145/2699276.2699290

2 Related Works

Our methods for procedural window effects span the areas of procedural modeling and building rendering. We review related works focused on rendering window light, building interior generation, procedural facade modeling, and interior mapping.

Rendering Window Light In offline rendering, window lighting is often created by placing light sources in the scene near windows (e.g. Mincinopschi [2008]). This requires that interior geometry exists for reflecting these lights. For real-time rendering, window lighting is usually simulated with proxy light sources. For example, Ahearn [2014] describes how to create an alpha channel for window textures to mask window areas that should be illuminated. Another method for simulating window light is to create an overlay texture for the lighted portion of the window (e.g. Young [2009]). In the area of procedural lighting design, Schwarz and Wonka [2014] use procedural techniques to place light sources on and around buildings using a set of goals for the desired exterior lighting. Our method provides a way to procedurally light windows on a building in a random style without the need to place actual light sources in the scene. To our knowledge, this is the first algorithm to procedurally simulate the window lighting in urban night scenes in a light-weight, controllable way.

Building Interior Generation Procedural building interior generation methods usually rely on building shapes and associated semantics to apply grammar, subdivision or graph layout approaches [Smelik et al. 2014]. For example, Peng et al. [2014] present an algorithm for subdividing outline shapes using deformable templates which can be used to create room divisions in a floor plan. Merrell et al. [2010] generate floor plans for residential buildings given constraints such as number and type of rooms. In contrast to these methods, our procedural room placement approach assumes no building size and shape. The floor plan is generated on-the-fly in texture space based on facade parameterization.

Procedural Facade Modeling Procedural modeling of facades aims to automatically generate variations in the appearance of building exteriors with minimal user interaction. Smelik et al. [2014] give an overview of these techniques, which include using grammars to describe facade elements and creating facades based on images. Our method could be coupled with these techniques to provide additional diversification to the facades. In particular, our work is related to Krecklau and Kobbelt's method [2011], which uses a grammar that can be evaluated in a fragment shader to derive facade elements. It also uses interior mapping to view rooms through windows on the facade, based on explicit storage of facade size and rule data. In contrast, our method represents room or floor extents on the facade implicitly by texture parameterization. Instead of generating more varieties of facade combinations, we aim to support room separation on existing textured facades with no extra storage.

Interior Mapping Interior mapping is a parallax effect where a building interior can be viewed from different angles when drawing only building facade geometry [van Dongen 2008]. Interior mapping computes the intersection of the view ray with the planes of the room to determine an intersection point in world space which is then used to look up in a room texture. In the original method the building is parameterized in world space, generally with uniform spacing for rooms. The drawback of this approach is that a world space parameterization of the building requires knowledge of the dimensions of each building, and room spacing must be set individually for each building. For a large city environment, generating and storing this individual parameterization in world space may not be feasible. Our approach to parameterizing rooms on a building facade using texture coordinates is much more flexible and can easily be used with interior mapping.

3 Facade Parameterization and Window Lighting

In this section we present two facade parameterization schemes, which allow us to use texture coordinates to define rooms (Section 3.1) and floors (Section 3.2). We propose methods to randomly light a subset of rooms in a scene using such parameterization schemes, with optionally varying lighting styles. The percentage of lit rooms is controlled using a threshold parameter in a continuous way.

3.1 Facade Parameterization Scheme

Our scheme directly uses vertical and horizontal texture coordinates to define room boundaries. We use the fact that hardware texture addressing (with `repeat` mode enabled) only uses fractional parts of the texture coordinates. Therefore we assume each integer range of texture coordinates defines the boundaries of a room, and each room in a given facade has a unique range of texture coordinates. For example, to define a facade with n by m uniformly distributed rooms we can draw a single quad and assign texture coordinates from 0 to n in one direction and 0 to m in the other direction and repeat the same texture for each room. Figure 1 shows examples of room placement using this scheme. Note that rooms can be placed in arbitrary locations with flexible combinations of sizes and shapes, as long as the windows belonging to each room are within the designated texture coordinate range.

To light a random set of rooms, we use a 2D function to compute a value between 0 and 1 for each room based on the integer parts of uv. We then compare the value returned from this function with the threshold value to determine if the room is lit or not. To avoid a harsh transition between lit and unlit, a smooth step function can be used to gradually turn the lighting on or off as the threshold value changes over time. To add variation per building, we can optionally add a third parameter to the lighting function which is a unique ID per building so that buildings using the same geometry will have different patterns of lit rooms.

Finally, in our texture scheme window locations are marked with an alpha value of zero. We combine the inverse of this alpha value with our computed room lighting so that room lighting only affects transparent window areas.

3.2 Pseudo-Random Function for Separating Rooms

The method described in the previous section allows for a precise definition of room boundaries. However, the drawback is that either the same texture is repeated for all rooms (Figure 1 left), or separate geometry has to be generated for each room or window (Figure 1 right). We found it more flexible to define a single texture covering an entire floor, repeating it vertically for the whole facade (Figure 2a). This greatly improves the variety of facade appearance while keeping the geometry simple. This scheme is commonly seen in GIS 3D building data. Note that this parameterization does not prevent us from repeating the texture horizontally or lining up multiple textures on each floor. The only change is that an integer range of the horizontal texture coordinate can now accommodate multiple rooms.

With this definition of floor boundaries, we introduce a pseudo-

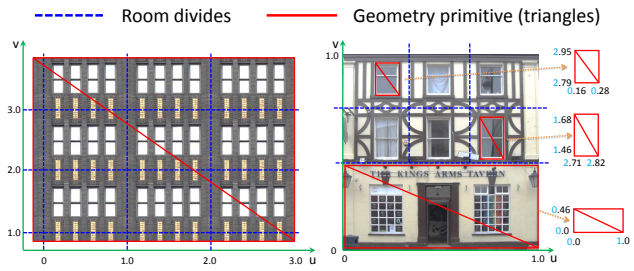


Figure 1: *Facade parameterization examples. Left: Uniform room distribution on a building facade, with each room occupying a unique integer range in both u and v, repeating the same texture. Right: Arbitrary room distribution on a building facade with a single texture. Each room has a unique integer uv range (highlighted in green), with the fractional part of uv representing the true texture coordinate. On the top two floors, only window areas have geometry and texture coordinates defined using our scheme. On the bottom floor, the entire room area is covered with a pair of triangles. Our scheme is flexible enough to accept these different ways of partitioning the facade.*

random window curtain function to generate variances of lighting among different windows on each floor. We evaluate this function per pixel using the horizontal texture coordinates and compare its value to a threshold value to determine if the pixel is lit or unlit. Like the previous method we also use a smooth step to blend between lit and unlit values and use the alpha channel to mark window pixels. Figure 2b shows a sample window curtain function and threshold.

The choice of pseudo-random function should not oscillate too many times between lit and unlit given the average number of windows in facade textures for the given models. We use the floor index to shift this pseudo-random function between floors to give a more random appearance to the pattern of lit and unlit windows. To avoid neighboring buildings having the same pattern of lit and unlit windows we can also pass in an (optional) index per facade texture that we use to further shift the pseudo-random function. For our models we chose the following function:

$$\begin{aligned} x &= \mathbf{t}_u + \text{floor}(\mathbf{t}_v) + b \\ f_{lit} &= \cos(2(\pi \sin(5x) + \sin(2x))) \\ lit &= \text{smoothstep}(thres - 0.2, thres, f_{lit}) \end{aligned} \tag{1}$$

where $\mathbf{t}_u$ and $\mathbf{t}_v$ are the u and v components of texture coordinate $\mathbf{t}$ and b is the optional building index. The value $thres$ is the threshold for controlling the percentage of lit windows.

Figure 2c shows the lit value computed for the facade in Figure 2a with $thres$ equals 0.4, and the computed window lighting is shown in Figure 2d. Note that this window curtain function creates a virtual separation of rooms within each floor. In Section 4.3 we will further demonstrate how to compute room separation walls using this function.

3.2.1 Varying Light Color and Brightness

To add more variety to the window lighting we use two different user-defined light colors. We use a second pseudo-random function to switch between these light colors. To avoid unrealistic sharp transition of colors, we also apply a smooth step here to blend between

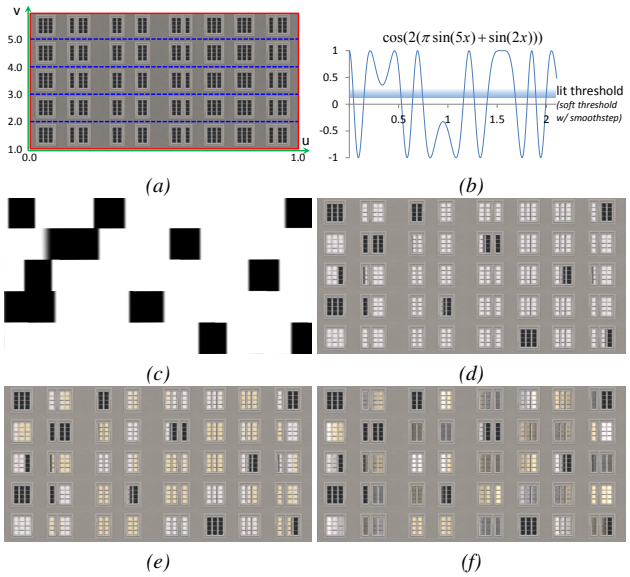


Figure 2: *Pseudo-random curtain and window light functions. (a) Facade parameterization with a single texture per floor. (b) Window curtain function and threshold. (c) Window curtain function applied to lighting on each floor. (d) Combined lighting and facade texture. (e) Lighting with two colors. (f) Adding varying brightness to window lighting.*

them. We chose the following function as our light color function:

$$\begin{aligned} y &= \text{floor}(\mathbf{t}_v) + b \\ f_{blend} &= \sin(19(\mathbf{t}_u + 10y)) + \sin(35\mathbf{t}_u + 10y) \\ \mathbf{c} &= \text{mix}(\mathbf{c_1}, \mathbf{c_2}, \text{smoothstep}(-0.3, 0.3, f_{blend})) \end{aligned} \tag{2}$$

where $\mathbf{c_1}$ and $\mathbf{c_2}$ are two alternative light colors, and $\mathbf{c}$ is the resulting window color. Note that we chose the two random functions f_{lit} and f_{blend} with different oscillation frequencies. We consider it natural to have color change in the middle of a lit window, since it simulates interaction of multiple light sources within the room.

In real city scenes rooms may contain lights of different brightness. To simulate this effect we use a third pseudo-random function to modulate the brightness of the light:

$$brightness = 0.8 + 0.4\cos(1.5(\pi \sin(7\mathbf{t}_u) + \sin(3x))). \tag{3}$$

The value of $brightness$ is in the range of 0.4 to 1.2 so it can either increase or decrease the amount of light.

4 Computing Room Locations and Interiors

Using the method described in the previous section for defining room and floor locations, we can extend this definition to 3D. In this section we present a method for obtaining world space coordinates of the rooms from the 2D texture coordinates. These world space coordinates are then combined with an interior mapping algorithm to render room interiors represented by a cube map. For our per-floor texture parameterization scheme (Section 3.2), we present a method for pseudo-random placement of room dividing walls based on our window curtain function.

4.1 Per-Fragment Room Localization Algorithm

Under our room definition described in Section 3.1, we can define a mapping between texture space and world space on a per-facade

basis, using the screen space derivative function available in fragment shaders. This can be used to find world space positions of the room boundaries. To evaluate the relationship between world space and texture space we use the fact that for the current pixel on the window we know both the world space position $\mathbf{p}$ and the texture coordinate $\mathbf{t}$. We can express the relationship between the current window pixel and another point on the facade with texture coordinate $\mathbf{t}'$ and unknown world space position $\mathbf{p}'$ using the screen space derivatives as follows:

$$\begin{aligned} \mathbf{t}' &= \mathbf{t} + r_x \mathrm{dx}(\mathbf{t}) + r_y \mathrm{dy}(\mathbf{t}) \\ \mathbf{p}' &= \mathbf{p} + r_x \mathrm{dx}(\mathbf{p}) + r_y \mathrm{dy}(\mathbf{p}) \end{aligned} \tag{4}$$

where dx and dy are the screen space derivative operators. We solve the first equation for r_x and r_y which we use in the second equation to compute $\mathbf{p}'$. Figure 3 illustrates this relationship between the screen space derivatives and the change in world space and texture space coordinates.

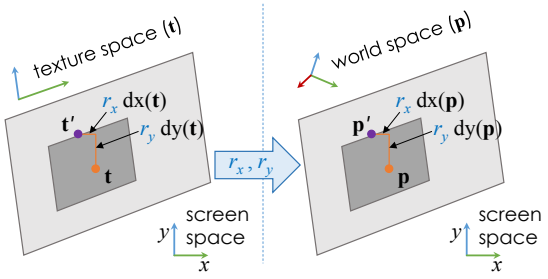


Figure 3: *When evaluating the window pixel located at the orange point, we can find the world space coordinates of another point (purple) on the facade by solving for r_x and r_y with known texture coordinates and then applying those ratios to the screen space derivatives of the world space positions.*

4.2 Solving Ray-Plane Intersections in Texture Space

For the purpose of interior mapping, we must compute the intersection of the walls, floor, and ceiling of a room with the viewing ray $\mathbf{v}$ in world space. To solve the ray-plane intersection equation we need a point on the plane $\mathbf{p}'$ and the normal to the plane $\mathbf{n}$. We use $\mathbf{p}$ as the origin of the view ray so that we can easily detect intersections that occur in front or behind the facade plane.

$$\begin{aligned} \mathbf{q} &= \mathbf{p}' - \mathbf{p} \\ t_{intersect} &= \frac{\mathbf{q} \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \end{aligned} \tag{5}$$

We can greatly simplify this equation with the observation that $\mathbf{q}$ is just $r_x \mathrm{dx}(\mathbf{p}) + r_y \mathrm{dy}(\mathbf{p})$, so there is no need to explicitly solve for $\mathbf{p}'$.

For each pair of parallel planes in the room we must find $\mathbf{q}$ and $\mathbf{n}$ so that we can compute $t_{intersect}$. For the floor, ceiling, and side walls we use the screen space derivative relationship established in Equation 4. For the back wall we assume that it is simply displaced from the facade along the facade normal $\mathbf{n}_w$ by an offset b_{off}. We define the following values for $\mathbf{q}$ for each of the planes:

$$\begin{aligned} \mathbf{d}_{fc} &= \frac{\mathrm{dx}(\mathbf{t})_u \mathrm{dy}(\mathbf{p}) - \mathrm{dy}(\mathbf{t})_u \mathrm{dx}(\mathbf{p})}{\mathrm{dx}(\mathbf{t})_u \mathrm{dy}(\mathbf{t})_v - \mathrm{dy}(\mathbf{t})_u \mathrm{dx}(\mathbf{t})_v} \\ \mathbf{d}_{sw} &= \frac{\mathrm{dx}(\mathbf{t})_v \mathrm{dy}(\mathbf{p}) - \mathrm{dy}(\mathbf{t})_v \mathrm{dx}(\mathbf{p})}{\mathrm{dx}(\mathbf{t})_v \mathrm{dy}(\mathbf{t})_u - \mathrm{dy}(\mathbf{t})_v \mathrm{dx}(\mathbf{t})_u} \\ \mathbf{q}_{fc} &= (n - \mathbf{t}_v)\mathbf{d}_{fc} \\ \mathbf{q}_{sw} &= (j - \mathbf{t}_u)\mathbf{d}_{sw} \\ \mathbf{q}_{bw} &= b_{off}\mathbf{n}_w \end{aligned} \tag{6}$$

where $\mathbf{q}_{fc}$, $\mathbf{q}_{sw}$, and $\mathbf{q}_{bw}$ are the $\mathbf{q}$ values for the floor/ceiling, side walls, and back wall respectively. The floor and ceiling planes have a constant y component of the texture coordinates n which is defined as either $\mathrm{floor}(\mathbf{t}_v)$ for the floor plane or $\mathrm{ceil}(\mathbf{t}_v)$ for the ceiling plane, and the side walls have a constant x component of the texture coordinates j which is explained in the following section.

We assume all of our buildings are pre-transformed such that the normal to the ceiling plane $\mathbf{n}_{fc}$ is $\langle 0, 1, 0 \rangle$. We already have the normal to the facade $\mathbf{n}_w$. We derive the normal to the side wall $\mathbf{n}_s$ as the cross product of the ceiling normal and the facade normal. We can directly plug all of these values into Equation 5 to find the intersection point between the view ray and each of the room planes.

4.3 Encoding Side Wall Locations Using Pseudo-Random Room Separation

For our per-floor texture parameterization scheme, our pseudo-random curtain function described in Section 3.2 only uses a loose definition of room separation. To apply interior mapping to these virtual rooms we must have a precise definition of wall boundaries. For this purpose we use the minimums of our pseudo-random function to define the boundaries of the rooms. By placing walls at the locations of the minimums of this function we can reduce the probability that the wall will appear in the middle of a lit window. Even if the wall is in the middle of a window, this portion of the window would generally not be lit except in the case where the lit threshold is set to 100%. Figure 5 shows an example of wall placement based on the pseudo-random function.

With irregularly spaced walls we need an efficient method to determine in which room a given texture coordinate lies. To do this we store the locations of the walls in a 1D texture such that given a horizontal texture coordinate, the locations of the side walls it falls between is returned from the texture look-up. For mobile platform compatibility we use an RGB 8 bit per channel texture to encode the walls, which means that we must scale our values to the zero to one range. Since our pseudo-random function has a period of 2π we first divide all of the wall locations by 2π. Then we iteratively fill in the values of the texture such that for a texture of size n the value at index i is the location of the walls w_j and w_{j+1} where $\frac{w_j}{2\pi} \leq \frac{i}{n}$ and $\frac{w_{j+1}}{2\pi} > \frac{i}{n}$. We store $\frac{w_j}{2\pi}$ in the red channel, $\frac{w_{j+1} - w_j}{2\pi}$ in the green channel, and the index of the room in the blue channel. Because our first wall location from the pseudo-random function is negative, we subtract the value of the first wall from all wall locations to ensure only positive values are stored in the texture. Figure 4 shows an example encoding of wall locations in a texture.

0	0	0	0.375	0.375	0.625	0.625	0.625
0.375	0.375	0.375	0.25	0.25	0.375	0.375	0.375
0	0	0	1	1	2	2	2

Figure 4: *In this simplified example the wall locations in texture space are shown in orange. In each texel the red channel is the location of the left wall, the green channel is the width of the room, and the blue channel is the index of the room. In our application we have a larger number of walls and use more samples to ensure high enough resolution in the shader.*

In the fragment shader we perform the reverse of this encoding operation to get back the location of the walls in texture space. We use the retrieved wall locations in the equations described previously

Figure 5: *Room placement and interior mapping based on window curtain function. Left: Walls are placed at the locations of the minimums of the window curtain function. Middle: Window curtains have the positive side effect of hiding room transitions. Right: Window texture combined with interior and window curtain rendering.*

for computing the world space positions from the texture coordinates.

4.4 Interior Mapping

Once we have computed the ray intersections with the walls, floor, and ceiling we must now find the value of the closest non-negative ray parameter t and determine its relative location within the cube map. For each pair of parallel walls only one of the intersection values with the view ray will be positive since the ray origin is the point on the facade which lies within the room span. Therefore for each pair of parallel walls we can take the larger of the two intersection values. Now we are left with 3 intersection points, and the closest of these is the minimum since none are negative. This gives us the world space position of the intersection point as $\mathbf{p}+t\mathbf{v}$.

To perform the interior mapping we need to construct a ray from the center of the room to the intersection point to use as the look-up ray for the cube map texture. We do not have an explicit world space position for the center of the room, but we know that in cube map coordinates the center is $(0, 0, 0)$. We can now find the relative position of the intersection point in each dimension by finding its relative position between each pair of walls. Because cube map coordinates go from -1 to 1 we can rescale our current texture coordinates and additionally define the facade as having a z texture coordinate of -1 and the back wall with a z texture coordinate of 1. By reusing our ratios of texture coordinate space to world space we can avoid explicit conversions of coordinates to world space. We obtain the following equations for the cube map look-up ray:

$$\begin{aligned} cx &= \frac{1}{(w_{right} - w_{left})} \left(\frac{t\mathbf{v} \cdot \mathbf{d}_{sw}}{\|\mathbf{d}_{sw}\|} - (w_{left} - \mathbf{t}_u) \right) \\ cy &= \frac{t\mathbf{v}_y}{\mathbf{d}_{fc} \cdot \mathbf{n}_{fc}} - (\text{floor}(\mathbf{t}_v) - \mathbf{t}_v) \\ cz &= \frac{t\mathbf{v} \cdot \mathbf{n}_w}{b_{off}} \end{aligned} \tag{7}$$

using the same definitions as in Equation 6. We can then use the vector $2\langle cx, cy, cz\rangle - 1$ as the direction for the look-up in the cube map.

4.4.1 Adding Variety to Rooms

When using a small number of cube map textures we can add additional variety to the interiors to increase the appearance of randomness. We combine the room index loaded from the wall location texture with the floor ID $\text{floor}(\mathbf{t}_v)$ to form a unique room ID for each room on a facade. We then use this ID to randomly flip front and back walls, given that a cube map has 6 faces but with interior mapping only 5 of them will ever be seen. We also use this ID to randomly flip between two light colors $\mathbf{c}_1$ and $\mathbf{c}_2$ on a per-room basis instead of using Equation 2 to blend them. Similar to the original interior mapping technique, we also introduce a furniture plane parallel to the back wall, by randomly selecting furniture textures from a texture atlas.

Because our wall placement scheme allows for arbitrarily placed walls, our rooms end up with different aspect ratios for the back wall. A simple way to avoid apparent texture stretching in this case is to avoid having all features with recognizable shapes in back wall textures, instead putting them in furniture textures, which are not stretched. Alternatively, since the aspect ratio of each room can be easily computed in the shader, we can use it to dynamically select from different cube maps or front and back textures, which are designed for viewing at different aspect ratios.

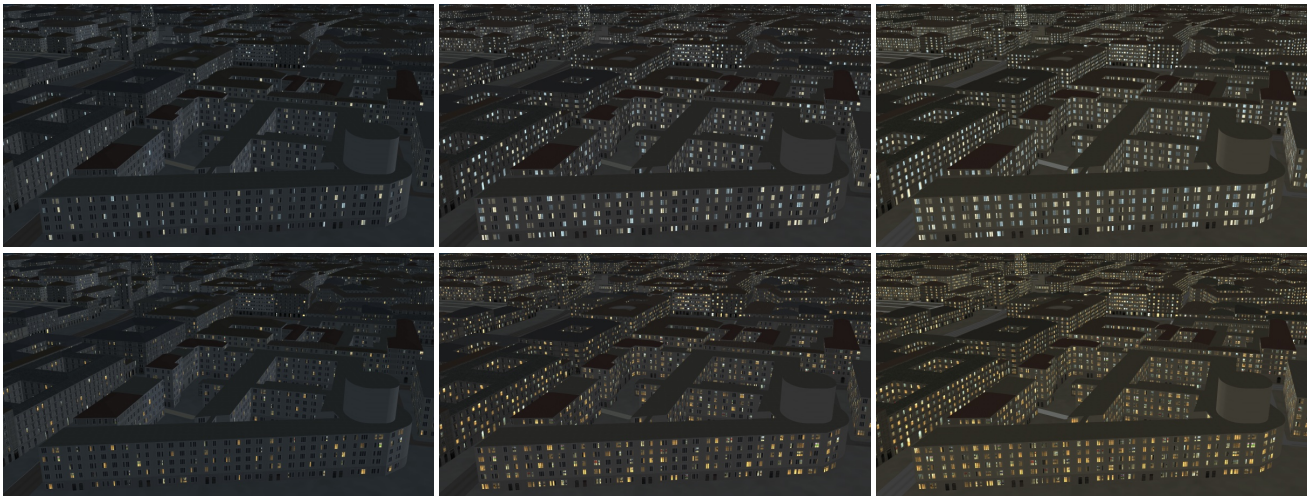
5 Results

We implemented our window lighting algorithms on our OpenGL 4.2 and OpenGL|ES 2.0 geo-visualization prototyping platform. All of our run-time algorithms are encapsulated in a fragment shader function that takes the texture coordinates and world space position of the fragment as inputs. We made our interior mapping algorithm optional in the implementation in order to suit the needs of different applications and platforms. We tested our implementation on both a desktop platform and a mobile platform. The desktop version was tested on an NVIDIA GeForce GT530 graphics card with screen resolution 1600×900. The mobile version was tested on a Google Nexus 10 tablet equipped with a Mali T604 GPU running Android 4.4.2, with backbuffer resolution 1280×720.

We use a forward renderer combining shading computations and our window lighting function. Since our window lighting function operates in screen space, it also works for deferred renderers, as long as texture coordinates, world space position and window alpha masks are available.

We tested our method on a city model obtained from a major GIS data provider. The model contains 329K triangles, which represent around 1.6K buildings sharing 156 generic building facade textures. The texture coordinates of the models were already designed as described in our floor definition scheme (Section 3.2). Therefore we did not have to convert or adapt the data. For the entire city we use three sets of interior textures and one 4×4 furniture atlas texture.

Figure 6 shows a comparison of window lighting and interior mapping effects with various window light percentages. In this overview perspective, our window lighting method generates two color lighting on a random set of windows and provides a convincing rendering of a city night scene. Our interior mapping extension further adds to the variety of window appearances. As mentioned earlier the *thres* parameter controls the portion of lit windows. In our supplemental video we show a time-lapse of the appearance change when adjusting this parameter continuously. Note that the transition resembles the effect of sliding the window curtains. This is not noticeable when the transition happens slowly, and is more



$thres = 0.2$ $thres = 0.5$ $thres = 0.8$

Figure 6: *City overview rendering with window light (top row) and interior mapping (bottom row) enabled. The three columns show different percentages of lit windows controlled by thres. The time-lapse showing continuous change of this value can be seen in our supplemental video.*

favorable than abruptly turning individual lights on and off, which may introduce distractions in many applications.

Figure 7 shows more close up views with interior mapping. It also shows that our method works well with complex building shapes. Since we infer room locations solely from texture coordinates, our method can be applied to any combination of planar facades, as long as windows do not cross texture boundaries.

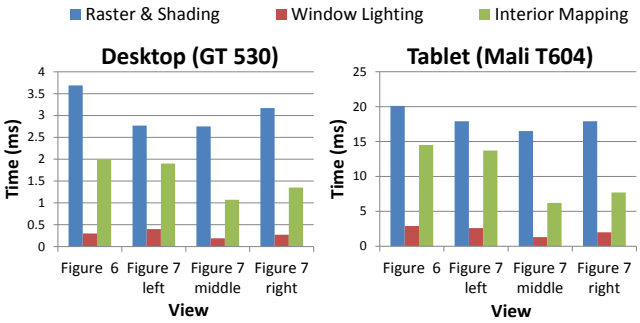


Figure 8: *Frame time breakdown of our prototype implementation running on desktop and mobile platforms.*

Figure 8 compares the performance of our prototype implementation from different viewpoints. We report extra frame time introduced by our methods by taking the difference of captured frame time with and without our methods. Our current shader code was written in a style that favors readability rather than efficiency, hence not highly optimized. Nevertheless, both window lighting and interior mapping run in a fraction of a frame time on our desktop (2011 low-end desktop GPU) and mobile (2012 high-end mobile GPU) platforms. Also, note that in our forward renderer we do not employ sophisticated view frustum and visibility culling or sorting methods, therefore our reported performance is tied with the depth complexity and overdraw of each viewing angle, as well as micropolygon rasterization inefficiencies. In a deferred implementation our method will be purely screen space. The performance would then only be relevant to the percentage of pixels that are marked as windows on the screen.

5.1 Limitations and Future Work

Our current interior mapping implementation does not enforce side walls to be placed exactly at left or right ends of the facade. With most buildings this is hidden behind an exterior wall so the extended room portion beyond the window area appears reasonable. With a small amount of extra computation in the shader one can rescale the curtain function per floor, so that end walls can be correctly placed. Alternatively, the wall locations texture can be edited in a preprocessing step to include walls at integral texture coordinates as well as the minimums of the pseudo-random function. In addition, like in the original interior mapping paper, our method does not ensure that rooms on building corners have the same interior when viewed from different sides of the building.

Our interior mapping extension works well on buildings composed of a series of planar facades, regardless of whether the overall building shape is convex or concave. However, it does not work well on window panes that cross the boundary between two polygons facing different directions. This is sometimes seen on low-polygon curved facades, such as the round building shown in Figure 9. The reason is that the screen space derivatives of $\mathbf{t}$ and $\mathbf{p}$ of two faces make a sudden change at the boundary, leading to a visible seam in the result. Fortunately this artifact is usually hidden behind window frames and walls, and even when seen it is not very distracting, given that interior maps are generally not used to render primary details. To eliminate this, one can either subdivide the surface to have smoothed geometry, or make sure that the transitions coincide with wall locations in the texture. Given that the majority of buildings in a typical city are composed of rectilinear sections, we feel that this limitation does not detract too much from our method.

We did not explore different options in furniture placing and combination in large rooms, since we feel that this direction is orthogonal to our topic. In the shader one can obtain the length and position of the room, as well as a random room ID. With these data more sophisticated algorithms can be employed to generate more variations of interior appearance. Also, with diverse interior designs, our pseudo-random room separation scheme may reveal room interiors of different styles through the same window, if the window is wider than the span of a curtain. This can be avoided by enforcing a minimum width in the curtain function, or switching to the more

Figure 7: *Additional views with interior mapping.*

Figure 9: *Our room localization and interior mapping algorithm does not work well on low-polygon curved facades, where visible seams may appear at polygon boundaries. However, this is not easily noticeable behind window structures and curtains.*

deterministic per-room parameterization scheme.

Finally, like all procedural shader functions, in theory our window curtain effect may lead to aliasing when minified or viewed at a glancing angle. We currently limit the width of smooth step in Equation 1 and Equation 2 to be no smaller than a constant factor multiplying the screen space derivative sum of $\mathbf{t}_u$, available in GLSL as the `fwidth` function, and it worked well in our case. More sophisticated methods can be derived to guarantee an appropriate screen-space frequency at transition bands.

6 Conclusion

We presented a new method to procedurally render random window lighting and interior details on buildings. Our method is purely fragment shader based and can be easily integrated into existing forward or deferred renderers. The window lighting component is very light-weight and runs smoothly on mobile platforms. The interior extension can be selectively applied statically or dynamically based on level of detail needs. As compared to previous methods, our scheme does not require precomputing or storing extra data per building and can be directly applied to buildings of different shapes and internal structures. Since our method is essentially an image-based approach, it is natively scalable to handle large city data without affecting performance.

Acknowledgements

This work was supported in part by an NSF Graduate Research Fellowship DGE-1148897. We also wish to thank Danxia Wang and Alexander Makarov for their design support.

References

AHEARN, L. 2014. *3D Game Textures: Create Professional Game Art Using Photoshop*. Focal Press, Burlington, MA, USA.

KRECKLAU, L., AND KOBBELT, L. 2011. Realtime compositing of procedural facade textures on the GPU. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XXXVIII-5/W16*, 177–184.

MERRELL, P., SCHKUFZA, E., AND KOLTUN, V. 2010. Computer-generated residential building layouts. *ACM Trans. Graph. 29*, 6 (Dec.), 181:1–181:12.

MINCINOPSCHI, A., 2008. Rendering an exterior at night in 5 simple steps, using VRay, Dec. `http://www.cgdigest.com/night-rendering-tutorial-vray/`.

MUSIALSKI, P., WONKA, P., ALIAGA, D. G., WIMMER, M., VAN GOOL, L., AND PURGATHOFER, W. 2013. A survey of urban reconstruction. *Computer Graphics Forum 32*, 6 (Sept.), 146–177.

PENG, C.-H., YANG, Y.-L., AND WONKA, P. 2014. Computing layouts with deformable templates. *ACM Trans. Graph. 33*, 4 (July), 99:1–99:11.

SCHWARZ, M., AND WONKA, P. 2014. Procedural design of exterior lighting for buildings with complex constraints. *ACM Trans. Graph. 33*, 5 (Sept.), 166:1–166:16.

SMELIK, R. M., TUTENEL, T., BIDARRA, R., AND BENES, B. 2014. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum 33*, 6, 31–50.

VAN DONGEN, J. 2008. Interior mapping: A new technique for rendering realistic buildings. In *Computer Graphics International 2008 Conference Proceedings*.

YOUNG, S., 2009. Procedural city, part 2: building textures, Apr. `http://www.shamusyoung.com/twentysidedtale/?p=2954`.