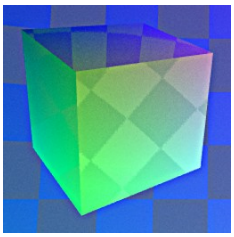
CUDA realtime rendering engine



Description	Basic realtime CUDA rendering engine with relfective surfaces.
Period	Spring 2013
Languages & Libs	C++, CUDA, SDL
Tags	Rendering

So far I've written a basic rendering engine which uses Nvidia's CUDA (Compute Unified Device Architecture) which can render reflective surfaces with environmental mapping and anti-aliasing and motion blur at 200 fps with minimal usage of 3rd party libraries such as OpenGL. This let me fully implement the cross-platform rendering pipeline from data transfer to pixel-level RGB calculations, all in C-like syntax.

An example rendered frame can be seen in Figure 1. Cubes aren't textured, but that would be fairly easy to implement since CUDA supports filtered texture lookups. However I'm likely to implement procedural volumetric textures, because the gameplay lets the user to arbitrary slice the object by drawing the cutting plane by mouse. If the objects were textured, then I'd need to implement also generation of UV coordinates and texture bitmaps.

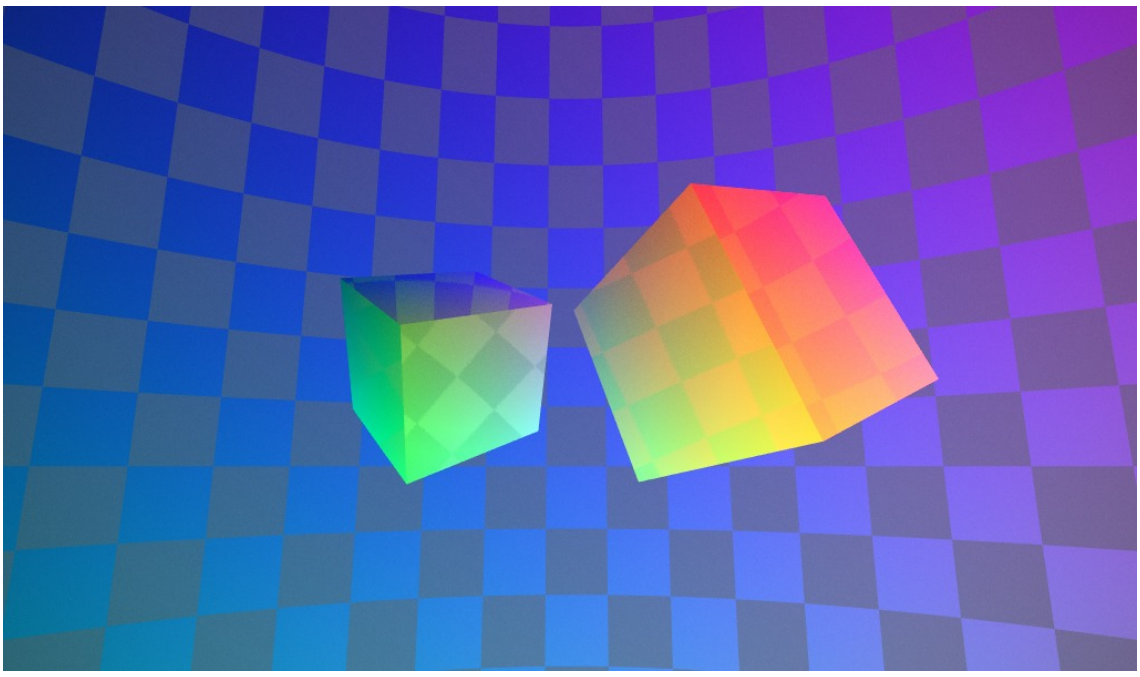


Figure 1: Example rendering of two cubes having reflective un-even surfaces, with antialiasing and vignetting. Current implementation is unable to make objects visible in surface reflections.

The antialiasing is implemented by having the CUDA thread to supersample each pixel, and store the averaged result. These results can be seen in Figure 2. In addition the used sampling pattern is mirrored between frames and the image is averaged with the previous image, so in practice the 4x sampling produces results which are similar to a more computationally demanding 8x sampling. This has the additional advantage of having the possibility of interpolating the sub-sampling direction based on camera motion since the previous frame to simulate motion blur on the background. This is demonstrated in Figure 3. This combined with very high framerate results in a very smooth user experience.

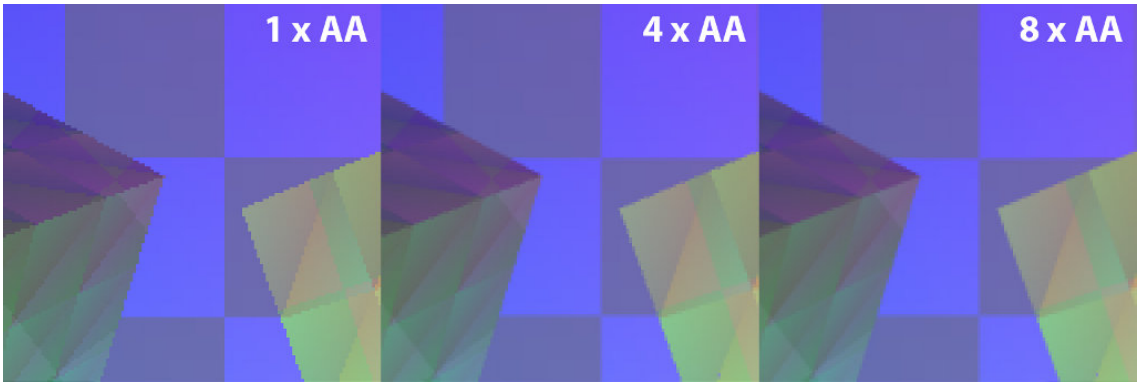


Figure 2: Displaying the spatial antialiased rendering with 1x (without), 4x and 8x supersampling.

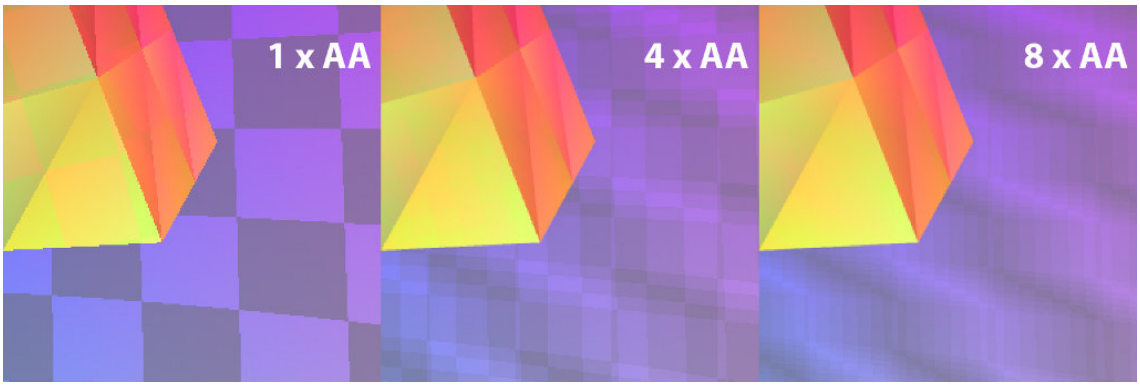


Figure 3: Displaying the spatio-temporal antialiased rendering with 1x, 4x and 8x supersampling.

CUDA threads operate in blocks of $16 \times 16 = 256$ threads in total, one thread / screen pixel. If the output resolution is 1280×720, there are $\frac{1280}{16} \times \frac{720}{16} = 80 \times 45 = 3600$ blocks to render. The number of active blocks depends on the hardware, but the used hardware (Nvidia GeForce GT 650M, maximum of 2048 threads) achieved a high occypancy quite easily. This is illustraded in Figure 4. If the thread block has to check ray-triangle collisions with only a single triangle, then it is encoded in blue, otherwise in red. Interestingly this can be solved quite efficiently in CPU as a pre-step for the rendering, by projecting the triangle corners to screen pixels and doing the checks purely in 2D.

If the models consist of large polygons, then each thread block needs to do only a few ray-triangle intersection tests. Also if that block does not contain any objects, the block doesn't need to render anything but the background, either by procedural calculations or by a simple texture lookup. The rendering is implemented as first rendering just plain background for the whole screen, and then have second pass on the region which contains renderable objects.

Ideally the rendered output would be displayed at the screen directly using OpenGL, but unfourtunately I haven't been able to get this working yet. The current work-around is to copy the resulting image from GPU to a SDL surface (in CPU's RAM), and then use a surface blit operation to display it on the screen. This adds a bit extra overhead, but could be easily refactored by someone who is more familiar with CUDA/OpenGL integration, and there is even an article "OpenGL Interoperability with CUDA".

The slicing process can be seen in Figure 5. By specifying two points in the screen space, based on virtual camera's parameter it can be determined that in which direction in world coordinates the two points are pointing at. By taking a cross product of these vectors, we get the normal of the splitting plane. By knowing that the plane goes through the virtual camera's focal point, we have full knowledge of its equation. Then all vertices of the models can be checked if its endpoints are in different sides of the plane or not. If they are, the whole triangle is flagged as being affected by the splitting action.

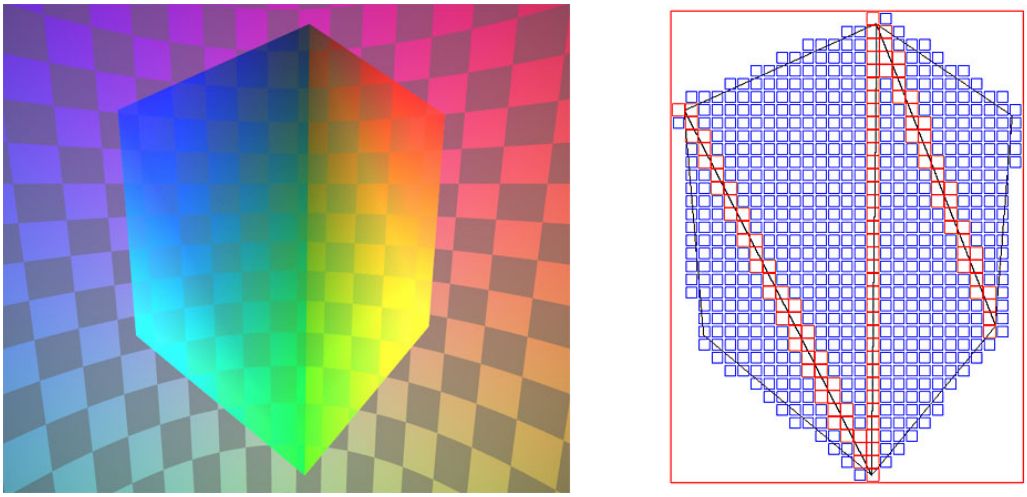


Figure 4: During rendering the screen is partitioned into independed 16×16 pixels blocks, which correspond to CUDA's thread blocks. Each block has 256 threads, one for each pixel. When antialiasing is used, each thread is iterated multiple times to determine the average color for the pixel. Pixels outside bounding box can be rendered with less effort, because only the background needs to be rendered.

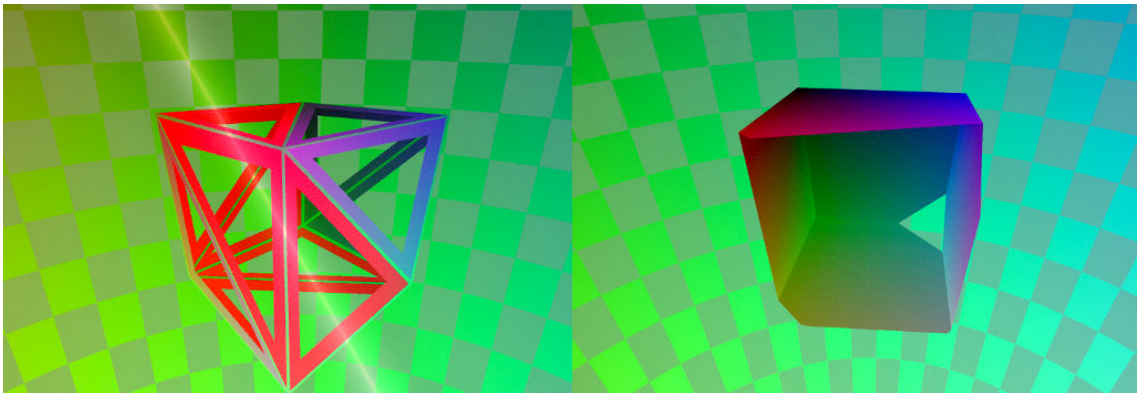


Figure 5: As shown on the left, the user can define a splitting plane, along which the object will be sliced and have its mesh refined. Affected triangles are highlighted in red. A result of multilple slicings is shown on the right. During the gameplay these holes would be automatically covered.

The actual splitting works by refining each triangle into three sub-triangles, one of which will be deleted by the splitting action. After all triangles have been refined, those edges which only have one linked triangle form the perimeter of the hole. At this point a proper triangulation needs to be automatically generated to fill the hole, and this should try to maximize the minimum angle of all resulting triangles.

The outcome of this algorithm is a valid mesh, but typically it has many redundant vertices which could be pruned. However so far I haven't been able to implement the procedure of refining the mesh while iterating through it, because data structures become temporarily inconsistent. Maybe an easier solution would be to copy the old structure into a temporary variable, and then re-create the optimized mesh from scratch. Without this optimization the mesh becomes unnecessarily complex after 4 to 6 splitting actions.