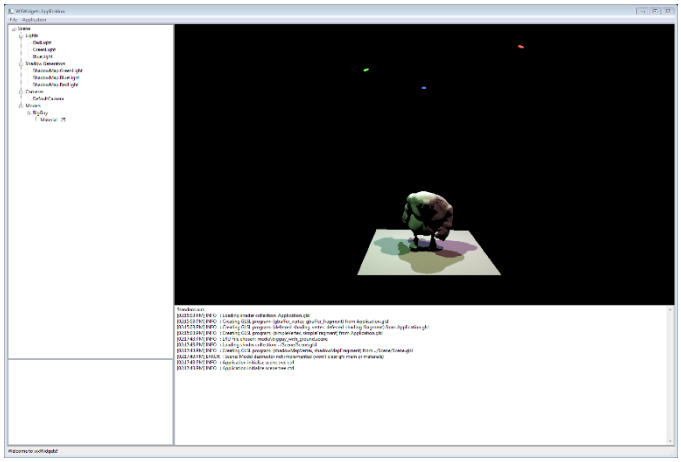
# Compact Procomputed Voxelized Shadows Demo

This demo application shows how to use the method described in “Compact Precomputed Voxelized Shadows” [Sintorn et al. 2014] to create an extremely compact data structure for looking up shadows from static objects. This document will quickly explain how the application works, and give an overview of the source code, while the source-code itself and the paper will hopefully explain the algorithms.

## Executable

**Requirements:**
NVIDIA GPU, 400 series or later.
CUDA 6.5 capable driver (latest driver will do fine)
64-bit windows
Visual Studio 2013 x64 Update 3 redistributable installed

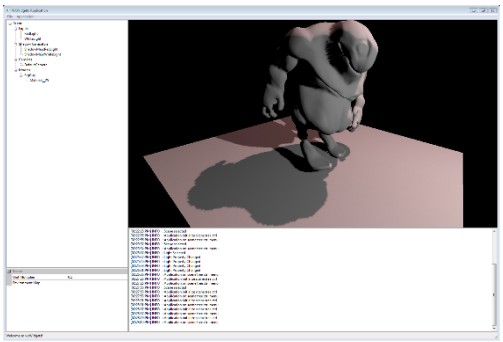
**Overview:**
Start the “CPVS2014/CPVS2014.exe” executable and you will (hopefully) be prompted to choose a recently used SCENE file (these are in a homegrown format used by the Chalmers Graphics Group). Press OK to load the accompanying scene.



You should now see the BIGGUY lit with three lights and simple shadow-map shadows. To the left is a TreeView describing the scene. Below the OpenGL view is a console with mostly uninteresting information.

You can navigate the scene using the WASD keys and your mouse. Lights and Cameras can be added to the scene by right clicking an item in the TreeView and choosing “Clone”. You will then need to give it a new name.

To position a light, the easiest way is to right-click the item and choosing “View and Control” and navigating it into place as though it were a camera. With a light selected in the TreeView you can also set its position, radiance and color in the property view below.



To get back to the main camera, right click it and choose “View and Control”. When you have created a new light, you can give it a shadow-map shadow-generator by right clicking it and choosing “Create Shadow Map”. In the screenshot to the right I have removed the green and blue lights and added a shadow-map to a white light.

Before creating a CPVS, right click the *shadowmap* that you want to convert in the TreeView and choose “View”.

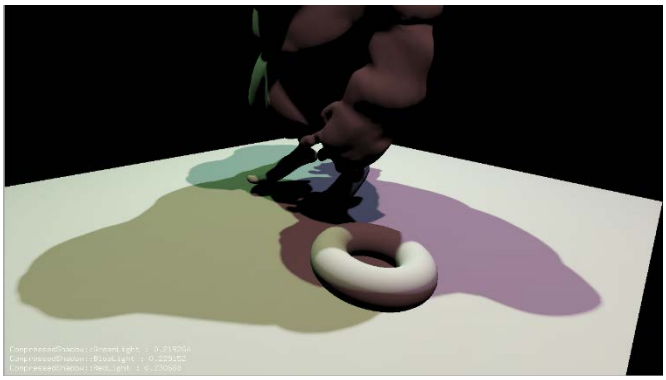
Then select the shadow map in the tree and adjust the FOV and aspect ratio until you have a reasonably good fit. Then adjust the Near and Far planes to fit the scene, without clipping it. This should be automatic of course, but... it’s not. Sorry. Assuming that your light is at about the same distance from the scene as the starting lights, a near plane of 10 and a far plane of 200 should be a good first guess (and you probably don’t need more precision).

Then, “View and Control” the camera again, select the shadow map in the TreeView and Choose “Application->Build CPVS...” in the menu. You will choose the resolution of your CPVS and whether the shadow-casters can be considered closed objects. In this case they can, so tick that box. Then press OK (choose a resolution < 16k so you don’t have to wait for too long).

You will then have to choose where to save the created data structure on disc, and wait till the program has finished building the datastructure (should only take a few seconds for a 16kx16kx16k data structure).

When done, you can see the size of the created data structure in the output, and the current meassured time for evaluating shadows from the data-structure is rendered in the GL view (both are circled in the screenshot to the left).

The “ShadowMap:RedLight” has been replaced by a “CompressedShadow:RedLight” and selecting that object, you can adjust the constant bias used and the current filtersize. Filtersizes between 2 and 8 will look weird in the current version (a low-prio bug in the new code) but filter sizes 1 or 9-17 should work well. You can create compressed shadows for all lights if you should want to.



Also, you can try adding a dynamic object to the scene. Choose “File->Import OBJ...” and choose the “torus.obj” file in the media folder. The torus should show up in the GL view and in the TreeView and by right-clicking it and choosing “Control” you will be able to move it around with the keys and mouse (then you will probably get confused and lose all sense of direction). The torus will receive shadows from the CPVSs, but will not cast any shadows unless you rebuild them.

You can save the current scene (File->Save or File->Save As..), and if you want to start from scratch with a new OBJ model, you can just cancel the first two dialogs upon starting the program, and import an OBJ of your choosing (the program may or may not crash if the OBJ lacks vertex normals, and it will ignore textures completely.)

# Source Code

**Requirements:**
Visual Studio 2013
CUDA 6.5

**Overview:**
There are five projects in the solution, but you only really have to care about one (CPVS2014). The others are:

- linmath – Linear algebra classes used throughout our projects.
- utils – A bunch of helper classes, most of which have proven useful enough to follow any project.
- scene – The (ever evolving) format we use for representing scenes. It's not complicated, and not interesting.
- CHAGApp – This is the base class (and support classes) I use for wxWidgets applications. It's constantly changing and quite a mess in many places, but hopefully you shall not have to look at it.

The CPVS2014 (Compact Precomputed Voxelized Shadows, 2014) project only contains a few files that do all the things you should be interested in:

- CPVSBuilder.cpp/ispc/glsl/h – The code that builds the data structure and saves to file.
- CompressedShadow.cpp/cu/h – The code that loads a compact data structure and renders shadows (for a GBuffer)
- main.cpp/Application.glsl – The main program, nothing very interesting here, and slightly cluttered with wxWidgets stuff.

This version of the code takes a different approach to building CPVSs (starting in **CPVSBuilder::build**). Instead of (as we did for the paper) bulding a complete sub SVO and the compressing this to a DAG through sorting, we now build the DAG immediately, by inserting new subnodes into a hash-map. We still build a small (e.g. 2048^3) subDAG at a time and combine them in a top DAG, but this is only because we cannot fit the entire shadow-map into memory.

We are currently working on a (much) improved building method which will be much faster, and moving to a hash-map was part of this move. The current implementation is not much faster than the original, but it is tremendously much easier to read, so we decided to distribute this version. Also, the new version has a much smaller working memory requirement (proportional to subDAG size, not subSVO size)

This version currently builds the DAG in a close-to-depth-first order, rather than the close-to-breadth-first order we used in the original code. We have not measured carefully, but have not seen that it has much impact on performance when rendering, but it's good to know if you start seeing suboptimal results.

The rendering code differs only slightly from the original version, the changes (if I recall them all) being:

- There is no antialiasing done at all – We decided not to move over the antialiasing code (VOAA or MRAA) as it makes things much harder to read, and is very simple to reimplement when you need it. If you should want any help with that, just holler and we'll squeeze it in.
- We only use a constant bias – In the paper-code, we calculated a "good" bias value in the fragment shader based on the dxdy of the light-NDC cords of the fragment. In the current code we only have a single constant bias. This is partly because it would be cumbersome to support several lights with the original code, and partly because you can probably figure out a better bias function yourself.

When building the code, make sure you set the CPVS2014->Properties->CUDA C/C++->Device->Code Generation appropriately. It's currently set to work on anything from GTX4xx cards, but for optimal performance on e.g. a Titan, you should set it to "compute_35, sm_35". Just a reminder.

Good luck! And send me an email if you have any problems (or successes for that matter).

Erik Sintorn – erik.sintorn@chalmers.se
Chalmers University of Technology