Prof. G. Zachmann
P. Lange

University of Bremen
School of Computer Science
CGVR Group
July 11, 2014

Summer Semester 2014

Assignment on Advanced Computer Graphics - Sheet 6

Due Date 24. 07. 2014

Exercise 1 (Procedural Brick Texture, *3 Credits*)

In the `Advanced Shader Techniques` lecture a procedural brick texture was introduced:

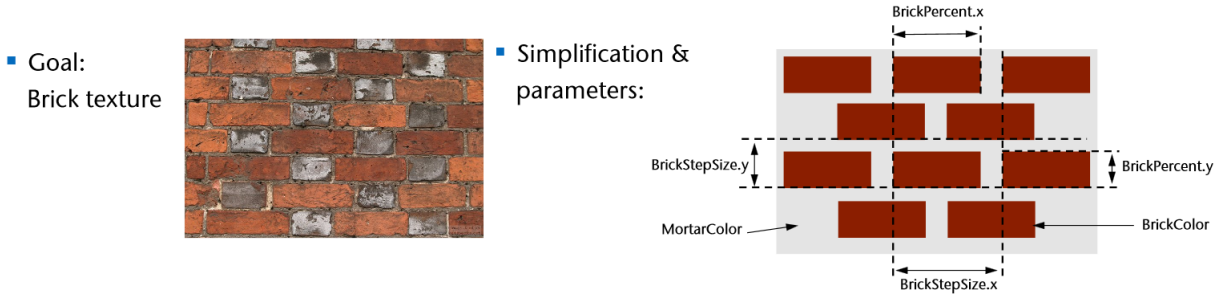

Figure 1: Procedural brick texture approach from the lecture.

Re-implement the idea in Ogre within the given framework:

- Use `BrickFramework::generateBrickTexture(double BrickStepSizeX, double BrickPercentX, double BrickStepSizeY, double BrickPercentY, Ogre::TexturePtr MortarColar, Ogre::TexturePtr BrickColor)` to generate a brick texture.
- Implement the functions `BrickFramework::generateMortar` and `BrickFramework::generateBrick`, which should generate the texture for the mortar as well as brick.
- For the brick and mortar texture use the `SimpleNoise` class or your favourite noise library to generate some interesting brickwalls. Also add some 'shadow' directly on the brick texture. Assume, that no lighting/shader/bump mapping/displacement mapping will create some shadows. See the example picture below for some reference of the bricks.
- The generated texture should be applied to the given `Ogre::Plane`, document your results with some screenshots.

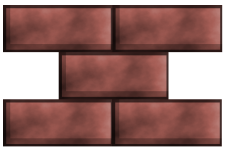

Figure 2: Simple brick example.

Exercise 2 (Simple Clustered Backface Tagging, *5 Credits*)

In the `Advanced Visibility Computations` lecture some methods for culling were introduced, one was the hierarchical clustered culling approach. Implement the following algorithm which should cluster a geometry and tag invisible triangles. The approach is as follows: First, it clusters the triangles of a given 3D model into a given number of smaller chunks. These chunks are organized in a hierarchical tree structure. In a second step, the visibility properties of the tree nodes are computed which determine the visibility of the geometry.

Your tasks in detail:

- Use `ClusterFramework::prepareModel()` to load and render the 3D model from disk. The `Ogre::MeshManager` can load the corresponding `.mesh` file with the `load` function. The `Ogre::EdgeData` with the normals, vertices and triangles can be retrieved via the resulting `Ogre::MeshPtr` from the read process. Draw only the triangle lines of the 3D model with the `ClusterFramework::draw3DLine( std::string name, Ogre::Vector3 from, Ogre::Vector3 to, std::string materialName)`. Render the original 3D model besides for comparison. Use this to check if you do not miss any triangles for your clustering process.
- Complete the simple tree structure `ClusterGeometry` which represents the hierarchical clusters of the geometry. Extend the given structure for your needs. It may be useful to add a list of triangles to `ClusterNode` and `ClusterLeaf`
- Cluster all triangles of the 3D model via implementing the `ClusterFramework::cluster( Ogre::EdgeData modelData, int amountOfClusters, Ogre::Vector3 cameraPosition, Ogre::Vector3 viewDirection)` function. It should create the hierarchical cluster of the geometry via using the `ClusterGeometry` class. The structure should have `amountOfClusters`. Therefore, you should try to equally distribute the `n` triangles of the mesh to the `amountOfClusters` clusters. The triangles should be inserted into the clusters with respect to the triangle position in z and camera position, expressed by the `cameraPosition` and `viewDirection`. Triangles which are close to the camera should be at the top of the tree structure, triangles which are far away from the camera should be at the bottom of the hierarchy. It may be useful to use a "layer-approach" which cuts the geometry in z direction in equal slices. These slices can be further splitted for better clustering (e.g. in x or y axis).
- Compute for each cluster if it is back-facing or not: `ClusterFramework::facing(ClusterNode *cn, double alpha, Ogre::Vector3 cameraPosition, Ogre::Vector3 viewDirection)`. If it is back-facing, use the `ClusterFramework::draw3DLine` function to colorize the triangle lines in red. For front-facing clusters, colorize the triangle lines green. It may be useful to use an angle α for a conservative set of triangles. You can use this angle when comparing the triangle normal with the camera view direction for better results. You should start with the root of the ClusterGeometry. Use the triangles visibility properties from parent `ClusterNodes` to derive visibility properties for child `ClusterNodes` instead of computing the visibility check for every triangle of every cluster.
- Determine (in theory) if your approach needs less visibility checks for the mesh when compared to an iterative approach, which would check every triangle. Argue, which cluster size `m` should be used for a mesh with `n` triangles when using your implemented clustering approach with visibility checks.
- Document your results with some screenshots.

Exercise 3 (Cube Maps, *2 Credits*)

Given the (s, t, r) texture coordinates of a vertex, assume that $|s|$ is the largest component by value, i.e., OpenGL will have to project (s, t, r) onto the side $x = 1$ of the unit cube (which is the parameter domain for cube maps).

Write down the calculations needed to compute the (u, v) pair on that cube side.