Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Oto Petřík

GPU Supported Terrain Editing

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: RNDr. Josef Pelikán
Study program: Computer Science
Specialization: Programming

2011

I would like to thank my supervisor RNDr. Josef Pelikán for valuable suggestions and patience. I would also like to thank my family for support during writing this thesis.

Contents

Název práce: GPU Supported Terrain Editing
Autor: Oto Petřík
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Josef Pelikán
e-mail vedoucího: pepca@cgg.mff.cuni.cz

Abstrakt: Vykreslování terénu je běžně prováděno na GPU. V této práci zkoumáme možnost využití tohoto dodatečného výpočetního výkonu k sestavení vlastních terénních dat. Spolu s přehledem běžných postupů editace terénu uvádíme také omezení na strukturu terénních dat, která jsou kladena různými způsoby zobrazení. Popisujeme počáteční představu reprezentace terénu, následné změny a výsledný návrh. Implementace návrhu byla provedena v podobě experimentální aplikace, která obsahuje sadu operací na úpravu terénu. Pro ověření rozšířitelnosti byl implementován efekt simulující erozi terénu. Na závěr nastíníme směry dalšího rozšíření editoru.
Klíčová slova: editace terénu, programování GPU, konstrukce shader programu

Title: GPU Supported Terrain Editing
Author: Oto Petřík
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: pepca@cgg.mff.cuni.cz

Abstract: Rendering terrain is routinely performed by GPU. In this work, we explore the possibility to use this additional computing power for assembling the actual terrain data. Limitations on terrain data structure, imposed by different terrain rendering methods, are presented along with an overview of common approaches to terrain editing. Description of initial idea for representing the terrain is given, including subsequent changes and final design. A experimental application implementing the design was written, containing multiple operations for terrain modification. To demonstrate extensibility of the architecture, an effect simulating terrain erosion was implemented. In the end, we outline ways to further extend the editor.
Keywords: terrain editing, GPU programming, shader program construction

Chapter 1

Introduction

Terrains are one of the very appealing objects of computer graphics, be it because of the ability to create virtual worlds or just because of interesting problems they present. Terrain rendering is used in fields spanning from scientific visualizations through film industry to games. Need for editing the data follows. There is need to manipulate gathered real world data, create backdrop for a movie or just provide gamers with piece of land to fight over.

Many papers have been written about rendering terrains using different methods and different hardware. Current solutions implemented in computer games allow for interactive frame rates, while still providing user with plausible images. This was largely made possible by introduction of faster GPUs and their increasing programmability.

1.1 Motivation

There are many tools that can be used to edit a terrain, ranging from basic image editors and map editors in games to complex 3D modeling suites. Although graphics cards are used to accelerate rendering in most of the editors, we believe that their potential in terrain editors is not fully utilized. Thanks to programmability of modern GPUs, there is a possibility to offload more work than just rendering.

1.2 Goals

Goals of this work are:

1. Study existing techniques for terrain editing.

2. Design a terrain-editing architecture based on programmable GPU with focus on extensibility.
3. Implement the system in an experimental application.

Chapter 2

Overview

2.1 Rendering geometry

While it is possible to render terrain exactly the same way as any other model in the scene, it is an ineffective way. Unlike most models, terrains have interesting properties that can be taken advantage of.

First of all, they are usually huge, with only a fraction of the area visible at the time. Resources are saved by splitting terrain into parts and rendering only the visible ones. However, not all parts are equal and further speedup is obtained by rendering parts with different level of detail.

Most approaches to terrain rendering use another property of terrains, they are generally flat. Although there can be complex structures, most of the terrain geometry can be described by a height function of terrain coordinates. Remaining geometry (caves, tunnels, overhangs, etc.) is considered separate and rendered as such.

There are additional problems involved with rendering terrain geometry (e.g. avoiding cracks between terrain parts, selecting proper levels of details, . . .), however from point of editing, it is sufficient to say that an editor must provide access to a heightfield (in corresponding level of detail) for every terrain part to be rendered.

2.2 Texturing the terrain

To render every pixel of resulting terrain image, renderer uses a lighting model. In addition to camera location, additional values are needed. These values (basic color, normal, etc.) are obtained from a set of textures or passed from the vertex program.

The simplest method to texture a terrain is to use a single set of textures (e.g. one texture containing color, one containing normal) and stretch the textures over the whole terrain. The upside is uniquely textured terrain without any repetitive patterns, downside is limited resolution of the textures (or prohibitive memory requirements). For rendering, it is possible to blend additional 'detail' texture to hide limited resolution or use virtual textures ([2],[3]) to work around graphics card memory limits. However, editing such data can be compared to using early graphics editors without support for layers.

Another method is to use a set of textures for every material, these textures are repeated many times over the terrain. Additional textures (masks) contain information where material is to be applied. Depending on the number of materials, rendering can happen in a single pass (all material textures and masks fit as fragment program parameters). In cases of terrain parts with many materials, multiple rendering passes might be needed, resulting color is calculated by blending partial results in the framebuffer.

Described approaches assumed, that the shader program is same over the whole terrain (all materials use the same shader). If materials use different shader programs, additional issues arise. Generic shader programs cannot be easily merged, while type and number of parameters passed from vertex program might be possible to unify, different fragment programs require different input textures (number, type or represented data) and different number of parameters.

One can render terrain in multiple passes, one pass for every material and blend results together in the framebuffer. Using geometry shader to filter geometry based on material mask, might be used to lower performance penalty. However, the limitation of combining materials in the framebuffer remains.

As seen in [1], another option is to assemble shader capable of evaluating every used material. In order to make merging material shaders possible, restriction upon shader code might be needed. For limited number of materials, such assembled shader might be all that is needed. Larger material sets might require assembling multiple shaders, each providing code required to evaluate all materials used in a given area. Upper bound of possible material combinations might be high, however it is not expected that number of materials used close together would be high. In case of large terrain, required shaders can be preassembled before rendering and cached.

While editing, camera usually moves at limited speed, this provides possibility to assemble shaders for newly visible terrain parts at runtime. Fast movement or large edit operations might require assembling higher number of shaders and cause decreased performance.

2.3 Editing

Offline Editing

The easiest way to edit a terrain is to modify the heightmap in an image editor. While this method is not usually feasible for correcting real world data, it can be used to generate heightmaps for games. Distributing materials can also be done in the image editor by painting masks to different layers and exporting them in a suitable format. Drawbacks of this approach are lack of feedback and limited size of terrains that can be edited in using this method.

Online Editing

To avoid feedback roundtrip between image editor and terrain renderer, editor and renderer can be merged. Complexity of the editor rises as image editor tools have to be implemented in way which does not interfere with the renderer (mostly with level of detail techniques). In the end, terrain editor transforms into sort of image editor which draws the image (and its layers) in quite different way. Work required for integration brings additional advantages, it is easier to add additional operations which might be required from the editor (entity placement, saving in file format used for final rendering, etc.). However main advantage is the elimination of roundtrip between editor and renderer.

Shader graph

In previous methods, we assumed that materials used in terrain rendering are already prepared - usually in the form of shader program with a set of textures and parameters. The work of editor was limited to painting material masks and heightmap.

Adding shader graph to the mix allows user to describe how terrain should look based on different input values (material textures, material masks, terrain height, slope, etc.). The graph describes fragment program in a user friendly way, editor must then assemble actual shader source used for rendering. Program can determine which parts of the graph are safe to prune and generate multiple faster shaders, with less required textures, for different parts of the terrain.

Constructing shader programs adds additional complexity to the editor, but allows user to further tune resulting terrain appearance. Using a graph to describe shader allows for additional flexibility. As seen in [1], combining with additional techniques (dynamic masks, sparse textures) makes it possible to allow limited editing in final game.

2.4 GPU vs. CPU views

In order to properly design terrain editing architecture, we look over different aspects involved in working with data for rendering and editing terrain from perspective of CPU and GPU. We limit the comparison to main points which affected the design.

Code running on the CPU has virtually unrestricted read-write access to the program's memory. This allows relatively easy manipulation of the terrain data. Downside is the limited number of threads that can be run simultaneously and requirement to share processor time. Given that CPU does not limit code structure the way GPU does, it is considered good practice to try offload processing to the GPU and make CPU available to parts of the program that cannot be easily run on the GPU.

GPU has different constraints, which additionally depend on used technologies, supported OpenGL extensions and GPU type. It is task oriented - a program is loaded, its dependencies are bound and the program is run. The result is either a picture in framebuffer, texture or list of values in a buffer. Accessing the result from GPU is fast, accessing the result from CPU requires transfer to main memory. Additionally there is a delay between starting a task on GPU and availability of the results.

Chapter 3

Design

3.1 Initial idea

From the beginning, the goal has been to offload as much processing as possible to the GPU while allowing user as much freedom as possible. It became clear that some variation of shader graph approach would be used. Changes in graph should be propagated instantly (except special cases of slow effects).

Initial idea was to consider terrain as stack of calculating stages. One stack for calculating terrain color, one stack for calculating terrain height and a stack for every additional terrain property (slope, sunlight, humidity, . . .).

Every stage of every stack is an operation. All operations, except constants and texture fetches, take result of lower level as input with possibility to connect results of other stacks as additional inputs. Output is considered result of the stack; at least until another stage is added.

Stages of the color stack would usually consist of applying one material after another, stages of the height stack would be applications of effects and merges of edit operations between effects. In order to calculate result of a stack, its dependencies must form a tree, this condition was later relaxed to directed acyclic graph. See figure 3.1 for example of such graph.

Create a new stack by calculating slope of height stack, add another stage to blend a texture, painted by user within the program, and use resulting stack as a mask in blending a material to the color stack. From editing standpoint it looks appealing, stacks represent understandable concepts. Assembling a shader program from such graph does not seem unreasonable.

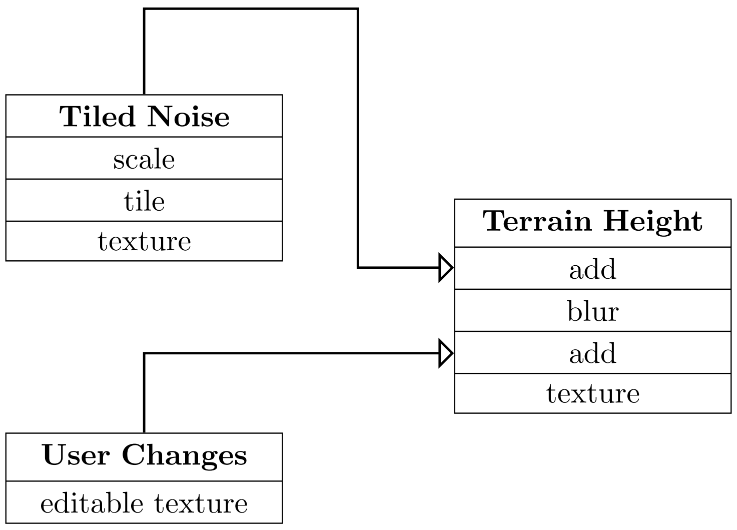

Figure 3.1: Example of a stack graph

Terrain rendering is to be done using an algorithm that avoids or minimizes cracks. For every part of terrain to be rendered, query consisting of part's location, size and level of detail is constructed. Resolving this query, graph provides a shader program, which is then used to render the part.

To optimize assembled shader, minima and maxima of outputs over areas are calculated and results are used to avoid generating shader code from parts of the graph; based on these values, stages have the option to behave as no-op. This is done to avoid running code and binding textures used for calculating a material in areas where material mask is zero. It would also lower the number of used textures in most versions of assembled shader.

3.2 Changes

The problem was dependencies, with every stack having only the top result, stacks would need to be split in order to allow access to values in the middle of the stack. Such access would be needed for example to calculate stack for erosion (which needs access to terrain height) which itself would be then accessed by terrain height.

Having output at every stage was considered. Every stage would have input from the previous stage, additional inputs and an output which would connect to the next stage and possibly used as input to different stacks. It would even allow stage to depend on results of multiple lower stages of the same stack. However, allowing outputs from within the stack would make graph more complex. Assembling a shader program from such graph, required rethinking of resulting structure of generated shaders.

It the end, it became clear that connection between neighboring stages of a stack is just another connection, resulting in a graph of freely connected nodes. While loosing the stack metaphor might leave graph a bit harder to navigate, it allows more freedom in constructing the graph. It is also makes easier to see direction of dependencies, making it easier to avoid circular ones.

Initial design called for two separate graphs, one for calculating height (used to assemble vertex program), another to calculate color (to assemble fragment program). However, some operations used in calculating color would need access to height data as well, to avoid duplication of height stack and its dependencies, the design was changed to single graph for generating both shader types.

Shader generation was also subject to changes. Every graph node can use information about target terrain area to provide suitable piece of shader code. This allows nodes to provide different code based on level of detail and terrain coordinates.

3.3 Final design

Terrain is represented as a directed acyclic graph of nodes called GENERATORS. Every generator has zero or more inputs and outputs parametrized by type of index (in current implementation always *float2*) and type of value (in current implementation type between *float* and *float4*). In order to use generator output, every input has to be connected. Additional generator properties can be set too, these are used to tune generator operation (texture filtering, constants used in shader, . . .).

Generators can provide EDITORS, which can modify operation of the renderer and receive user input. This is used in *editable texture generator* and allows modification of textures.

As result, terrain is editable in two different ways - one by modifying properties of generators, second by changing the graph that calculates terrain height and terrain color.

To render a part of terrain, the graph is evaluated for vertex and fragment program. Requests for areas required from given generator are merged and generator is then queried for smallest enclosing rectangle.

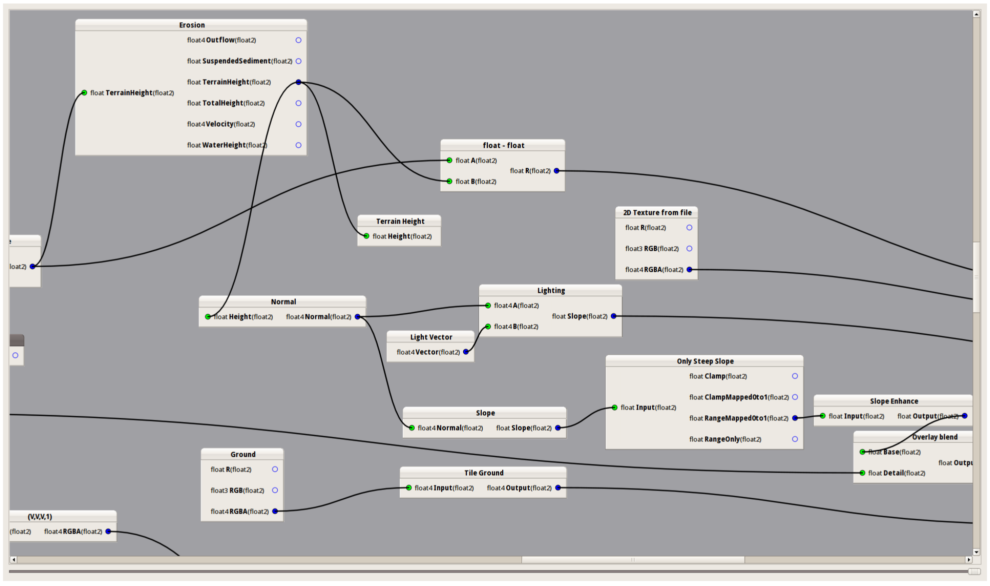

Figure 3.2: Part of terrain graph in the application

Generator is free to provide code, that uses any of generator's inputs, the area used to query inputs can be enlarged if needed. In order to avoid multiple instances of single generator, queries on input have the same level of detail as the query being evaluated.

It is also possible to evaluate a subgraph of inputs independently, use assembled programs to render input values to a texture and return a code to sample the texture, this is used in *render-to-texture generator*.

Generator outputs can also be queried for a minima and maxima over given area, this can be used to avoid evaluating unnecessary parts of the graph. It is also used to calculate height values for bounding boxes of terrain parts. See figure 3.2 for example of the graph in final design.

Chapter 4

Implementation

4.1 Used technologies

Program was developed on the Linux operating system. This slightly affected selected technologies. That said, libraries used are portable and a Windows build was created. Program is written in C++, with Qt4 for GUI and OpenGL & Cg for graphics. Other supporting libraries are glew, loki and openil. Libraries' homepages are listed in appendix Programmer's documentation, which also contains overview of the implementation at the source-code level. The application is present on included CD, see CD content for further information.

4.2 Data representation

Terrain description is stored in a graph of generators (nodes).

Each generator has a name, set of MODULES, named inputs and named outputs. Graph contains at least two generators with no outputs[1], which represent final terrain height and color. Additional such generators can be added as probes to allow easier access to terrain rendered using only a subgraph. Generators are assigned to layers and circular dependencies are prevented.

Aside from the main value, each generator output also provides a *res* value. Its estimate is accessible in main program, real value can be location dependent and is accessible only in generated shader. This value describes distance to neighboring texel; for generators without raster (e.g. constant color or value) a default value is returned. Providing *res* value

[1]The generators have outputs, however, they are hidden from the user

allows creating generators which calculate normal vector over the terrain.

Every time generator is queried, it resolves itself to one of its modules. Its responsibility is to provide Cg code representing the generator and also signal changes, which might require re-querying of the graph. Like generators, modules have inputs. They have the same signature as generator's inputs, however, not all generator inputs have to be present. Inputs not present in module are not queried, which allows for smaller shader program. Module inputs also contain *epsilon* value; query area of connected generator is incremented by *epsilon*[2] in every direction. This ensures safe sampling of neighboring texels.

Generated Cg sources are compiled and the result is cached.

Changes in the graph or generator properties are detected and new shaders are constructed.

4.3 Rendering

Terrain geometry is rendered using a simplified version of algorithm described in [1].

The terrain is split to square tiles of different sizes. Parts of the terrain which should be rendered in higher detail are covered with larger number of small tiles, lower detail parts are covered with few large tiles.

Each tile consist of 16×16 vertices. If all tiles are of the same size, triangulation is regular. If a side borders with larger tile, every second vertex is skipped; this is done to avoid terrain cracks. Additionally, tile neighbors can only be same size, nearest smaller or nearest bigger size.

In order to determine tile size, current implementation considers only distance from the camera. Tile size is also used to select level-of-detail, value which is further passed to shader program construction. To avoid rendering unnecessary geometry, an octree is used to cull parts of the terrain outside the view.

For every rendered tile both vertex and fragment programs are generated. Minimum and maximum height over the area is also calculated and used for frustum culling.

[2]Unit of *epsilon* is local *res* value.

4.4 Editing

Graph structure

Editing the structure of the graph is done through separate GUI window. Qt's GraphicsView is utilized to present user with editable graph of generators. Items representing generators draw inputs on their left side, outputs on the right side. Changes made in the window are mirrored to the graph used to generate shaders. User is prevented from connecting outputs of generator which does not have all inputs connected, creating circular dependencies is also prohibited.

Generator properties and data

Generator properties are set on the right side of the main window. Generator's editor is responsible for drawing that area and implementing changes. While most editors only set generator properties, editor can also intercept user input and render additional geometry. With transform feedback enabled, the terrain is rendered for second time, geometry shader is used to test triangle against camera-mouse ray; this avoids copying terrain data to the main memory and performing testing on the CPU.

Editable generator is an example of a generator that can edit data. Editor draws additional geometry (cursor) and processes user input.

Changes of generator properties trigger regeneration of shader programs if needed.

4.5 Encountered troubles

Using Cg combined with the need to render to a buffer object (for terrain-ray collisions) required using NVIDIA-only OpenGL extension [3], this prevents program from running on graphics cards without this extension.

Using Qt library for GUI worked well most of the time. Minor issue was encountered in initializing additional libraries (Cg, glew) at the right time. However, sharing OpenGL context with overlay text drawn by Qt in the main window turned out tricky[4]. In the end, it required carefully separating code which needs to access OpenGL and code which might even indirectly trigger repaint event.

[3]NV_transform_feedback

[4]For details, see GUI section of Programmer's documentation

Boost::serialization turned out to be a mixed blessing, it allowed fast initial implementation, but bugs caused by slight mishandling of the library were hard to diagnose.

Chapter 5

Extensibility

In order to test the extensibility of the program, a non-trivial generator was written. The implementation is based on the water erosion simulation described in [4].

Generator has single input (terrain height) and multiple outputs (terrain height, water height, total height, outflow and velocity).

At the beginning, the subgraph connected to the *terrain height* input is evaluated and the result rendered to the texture. Simulation is then run between two sets of three textures. Seven render-to-texture operations are needed for single simulation step, shaders used are hand written.[1]

Once simulation is finished, generator is marked as 'current' and a module sampling textures with simulation results is returned. Simulation is re-run if input subgraph is changed or generators within subgraph change their values. To control the simulation, an editor was written, it allows user to set simulation parameters and once satisfied with the values, re-run the simulation.

Resulting generator does return modified terrain height, however, tuning parameters to provide likable result is rather difficult.

[1]If a generator requires to use a generator input in such case, providing a subclass of `shadergen::Pass` class is needed

Chapter 6

Conclusion and future work

6.1 What went well

We designed and implemented an extensible program for terrain rendering which runs on both Linux and Windows. It is possible to extend the program by creating additional generator types[1]. Program is able to synthesize Cg shader programs based on graph description given by user. Changes in the graph structure or generator properties are reflected in timely manner in the rendered image.

6.2 What did not went well

Probably the most troubling was underestimation of the scope of the problem. Initial design silently expected that a variation of sparse virtual texture ([2],[3]) would be implemented as an early modification, this would allow larger terrains and effective caching. Delays caused by obtaining minimum and maximum of terrain height turned out to be higher than expected and visibly slow down program in cases of fast camera movement. Debugging the program was also harder than expected.

[1]See chapter *Extensibility*

6.3 Future work

Performance of the solution can be improved, adding sparse virtual tree implementation and effective caching of generator stats should visibly improve performance.

Extending the program to generate geometry programs would allow using terrain data for filtering geometry, it would allow for easy automatic placement of entities such as trees and rocks.

Last but not the least, for serious use, more tools for texture editing and additional generators would be needed.

Appendix A

CD content

Included CD-ROM contains developed application, directory list follows:

- `/thesis` - PDF version of this thesis
- `/bin-linux64` - compiled binaries for x86_64 Linux
- `/bin-win32` - compiled binaries for 32-bit Windows
- `/src` - source tree of the application
- `/src/texts` - user documentation (in Czech), also copied to directories with compiled binaries.
- `/src/html` - Doxygen generated documentation, includes a copy of Programmer's documentation.

Windows binaries folder contains required libraries. Linux binaries expect required libraries installed in usual paths[1]. For building from sources, see Programmer's documentation for details. In order to run the program, system with NVIDIA graphics card 8800 or later is needed. The program was developed on Linux operating system, while Windows binary is provided for convenience, it had received less testing than the Linux version. In case of problems running the application, author can be contacted at oto.petrik@gmail.com

[1]Binaries compiled on x86_64 Fedora 14

Appendix B

Programmer's documentation

This appendix contains modified version of programmer's documentation. Original version is incorporated by Doxygen in generated documentation stored doc/html subdirectory of the source tree. Following text has been modified for easier reading without Doxygen's links, however, occasional glance at the source code might be needed.

B.1 Introduction

This program allows for terrain editing using a graph of connected shader parts that compute height and color of the terrain. See texts/userdoc.txt in source directory for user documentation.

Program was developed on a x86_64 linux system, care has been taken to allow building under visual studio express, however windows-only bugs might still lurk somewhere.

Installation

Program requires compiled binary and directory 'cg' from source tree. Example file 'example.terr' is provided (which requires directory 'images' from source tree to be present). OpenGL 3.2 with NVIDIA transform feedback extension [1] is required to run (NVIDIA 8800 type card or newer).

[1] GL_NV_transform_feedback

B.2 Compiling

Program uses CMake build system. Run cmake in directory containing file CMakeLists.txt, then make (on linux and like) or nmake (windows). CMake can generate Visual Studio solutions. Program was developed using GNU g++ on Linux, Windows binary was created using Visual C++ 2008 Express. On Windows, FindDevIl.cmake file in CMake installation might need tweaking - find 'IL/il.h' instead 'il.h'.

Dependencies

Required libraries

- Qt 4.7 `http://qt.nokia.com/`
- DevIL `http://openil.sourceforge.net/`
- loki `http://loki-lib.sourceforge.net/`
- Boost[2] 1.44 `http://www.boost.org/`
- NVidia Cg 3 `http://developer.nvidia.com/cg-toolkit`
- GLew `http://glew.sourceforge.net/`

and CMake build system (`http://www.cmake.org/`) to compile the program.

Sharing OpenGL context with Qt library is sometimes tricky, if you use Qt version newer than recommended changes might be needed.

B.3 General notes

Coding style

Older parts of code use `lots_of_underscores_and_small_letters` style, which suffers from collisions between class names and variable names. This became problem only later in development, after some parts of code were already written using the style; it also lead to using multiple namespaces to minimize collisions. Some of the code was converted to the new style, unfortunately, parts of the source still use older style.

New style is camel-case based and does not suffer from collisions as much, although it has few glitches (naming enum values). See *shadergen* subdirectory for example of new style.

[2]For Windows build use version 1.43.

Used naming conventions

Methods named `attach_something`[3] (or `detach_something`) take (or give up) ownership of object passed as parameter. Names `register_something` (or `unregister_something`) are used for methods that do not take (or give up) ownership of the object in question. Unless stated otherwise, all registered objects have to be unregistered before destruction of the object. Although there are getters returning non-const objects, do not modify them unless absolutely sure. Usual case is modification of data before binding/registering/etc.

Removing object from a collection

If object A needs to unregister/detach/... itself from a object B that keeps a pointer to A, extra care has to be taken to make sure that given code is not called from a loop over pointers owned by B. Most collection-like classes use `std::set` or `std::list` that can withstand invalidation of neighboring iterator, however none of the classes can withstand invalidation of current iterator.

Use Eclipse's "Open Call Hierarchy" function or Doxygen caller graph feature to find out callers of the code in question and change caller's loop to either safely iterate using two iterators or to make a copy of the collection and iterate over the copy.

B.4 Generating shader programs

Shader program construction occurs in two phases: user creates a graph, program converts the graph to a shader.

Creating the graph

User connects GUI widgets representing generators to construct directed acyclic graph. Few basic rules have to be followed:

- no loops (checked using `Generator::dependsOn`)
- to connect to generator's output, all its inputs have to be connected (checked using `Generator::hasAllInputsConnected`)
- to remove generator, all its outputs have to be disconnected (checked using `Generator::hasSinks`)

[3]Old coding style convention is used because most affected classes still use it.

The rules make sure that it is always possible to convert graph to a correct shader. Exceptions to this rule are *probe generators*, they are created without connected inputs, but can be used as target generator for shader construction, see `Tree` members `currentVertexOutput` and `currentVertexOutput` for code that avoids using invalid probe generator as target.

Converting graph to shader

Graph is stored in a `Tree` (which is really directed acyclic graph, not a tree; idea evolved, name did not), target generator is selected using either one of `Tree's` methods to get default generator or another means (`RTTGenerator`). Generated shader is valid only over given area and provides data for given lod[4]. Using these parameters, one of `Pass` descendants is constructed.

For every generator, pass keeps requested area. Generators are required to provide a module (`Generator::provideModule`), module is used to determine requested area of generators connected to inputs. Generators are processed from higher layer to lower to make sure that all dependent generators (and provided modules) have opportunity to enlarge requested area (`GeneratorInput::requestedArea`). If the area is empty, no module is requested. This occurs if generator has multiple inputs and provided module has only some of them. It is also the core of shadergen usefulness, generated shader contains only what it needs — no extra textures or code that would not be used anyway.

Once pass is resolved, resulting shader and parameters can be obtained. Cg programs generated for vertex and fragment program share TEXUNITn semantics, to avoid collision Pass constructors have optional parameter to start counting TEXUNITn names where previous shader (vertex program) left.

For usage, see `Tile::provideFragmentProgram`.

Generated vertex program are also used for mouse cursor vs. terrain testing (see `terrain::intersection`).

Serialization

Serialization is done using `boost::serialization` see *Lower level classes* section for details.

Generators usually serialize all modules and modules serialize all inputs and outputs (samplers are not serializable and have to be recreated).

[4]an unsigned value selecting the level of detail

Some generators prefer to recreate all modules on deserialization, those change value of `Generator::serializeModuleList_` to avoid serialization of modules.

B.5 Terrain

Program displays terrain as a set of triangle meshes, with every mesh is rendered separately.

Terrain is subdivided to allow more detailed rendering where it is needed. Current version uses only distance from camera to determine which parts are important. If fast stats[5] caching is implemented, it would be possible to use differences in terrain heightmap (and between heightmap lods) as well.

Nodes

Internal representation of terrain is a collection of nodes (`terrain::node`) which can be accessed using `node_storage`. Every node is in one of two states:

- split (does no rendering, only keeps track of its children)
- leaf (does rendering using `terrain::Tile` class)

Every node can be selected using value of type `id_t` as an index.

There are rules to ensure terrain without cracks, the most important is that every neighboring pair of nodes differs in `id_t::lod` by zero or one. Lod value zero means that node is as detailed node possible (no vertices are skipped).

If lod does not match, node with *lower* lod uses irregular mesh to avoid cracks.

Changes between node states are propagated to neighboring nodes, which allows them to update their meshes accordingly.

Tiles

Leaf node uses `terrain::Tile` class to render the mesh. Tile class requests vertex and index buffers from `tilegen`. Vertex buffer is shared by

[5]`shadergen::GeneratorOutputStats`

all nodes, index buffers are shared by all nodes that have same `neighbour_bits` signature. If `node::neighbour_bits` change, method `updateBorders` is called to refresh the index buffer.

Tile uses stats to calculate bounding box. `TerrainFragmentPass` and `TerrainVertexPass` are used to construct shaders for displacing and coloring the heightmap. Invalidation of stats, fragment or vertex program is detected and tile is registered with terrain object for revalidation.

B.6 GUI

Main windows and OpenGL

User interface for editing the generator graph is implemented in *TreeEditScene.hpp* and *TreeEditScene.cpp.*

CustomGLContext.cpp contains GLEW, Cg and OpenIL initialization. OpenGL context is shared between terrain rendering and normal QGraphicsView rendering [6].

Precise moment of GL context initialization seems to vary between Qt platforms, `GraphicsView` (derived from QGraphicsView) takes extra care to render only after all libraries have been properly initialized.

To avoid GL state conflict, GraphicsView and other parts of the GUI use `TAKEOVER_GL` macro to save Qt's OpenGL state, set GL to known state and enable Cg. The macro has a counter making it safe to have multiple instances on the stack. Whenever Qt event handler or slot requires to do something with terrain, shadergen or perform an OpenGL call, it must use the macro to avoid fighting with Qt over GL state.

Editors

Editors are classes, usually derived from QWidget, which allow changing generator parameters (filtering, changing constant values, etc.) and/or editing generator data (painting into texture).

`Generator::editorName` holds type of editor required to modify given generator, using this name and a builder system, `EditorSelectionWidget` creates new instance of required editor and generator keeps this object for its lifetime (`Generator::setEditorObject`).

`Editor::activate` and `Editor::deactivate` are used by editors to temporary override default vertex and/or fragment target generators (us-

[6]For details see code in *app/GraphicsView.cpp*

ing methods `overrideVertexOutput` and `overrideFragmentOutput` of `Tree` class).

`Editor::terrainTest` is called with information whether mouse cursor is over the terrain and if so, terrain coordinates of the cursor. This is used for texture painting (`EditableTexture2DGenerator`).

B.7 Lower level classes

Serialization

Serialization is done using the boost::serialization, resulting file is an XML file with an unfortunate structure. The library does not support marking which pointers should be used to serialize only references to already (or later) serialized object and which pointers should serialize objects themselves. However, an XML file is still easier to debug that binary file (the other boost::serialization option).

See boost documentation on details how the library works. Relevant files that show how the library is used are *serialization.cpp*, *Generator.hpp* and derived generators.

It should be noted, that using serialization might produce warnings about unused variables. It also significantly slows down the compilation.

File serialization.cpp contains functions that allow mapping absolute filenames to relative, this allows deserialization from different absolute path to the one used in serialization.

OpenGL & Cg

Wrapping of OpenGL and Cg libraries is done in *src/gl* directory. Most of it is straightforward, interesting parts are probably *util/shaders.cpp* for loading shaders from *cg* subdirectory of program directory and *buffer_bindings.hpp*, *buffer_set.hpp* and *buffer_targets.hpp* which together allow reasonable management of buffer object for indices, vertices and vertex attributes.

Render subsystem

Render subsystem orders drawing calls to make sure all instances of a model are drawn together without unnecessary state changes. Rendering order of models is arbitrary, there could be a speed gain in ordering models to minimize state changes between models, however current program has relatively few models in the view at a time.

For now, render subsystem serves mostly to avoid mixing rendering tiles and rendering bounding boxes (if enabled).

Camera classes (e.g. `render::generic_camera`) allow for coupling render instances with matrices required for rendering.

Scene management

Scene management is done using an octree. Class `scene` has views (each with a camera). Every `scene_view` tracks visible nodes and entities[7].

Method `update_entity` of `scene_view` class handles updating render instances which should be rendered using given camera. There is no mapping between `tree_entity` and `render::instance`; `tree_entity` is free to add multiple `render::instances` to camera in the `show` method. Only requirement is to remove those instances in `hide` method.

There is a debug feature to allow displaying bounding boxes of tree nodes. It is expected that bounding boxes for entities will be drawn by entities themselves if required.

Image

Files *Image.hpp* and *Image.cpp* containing wrapper class over OpenIL library. It is used to store and load textures. Supported types are RGB8, RGBA8 and 16bit grayscale image (useful for heightmap).

Builders

Given that generators select their editors by a string, there is a need for system that will, for given name, provide a configured instance of a class or an instance of derived class. These mappings can be spread over many files (see number of editors and generators), requiring an initialization code to include all header files and register them would be error-prone. File *util/builder.hpp* solves both, it provides templates and macros, that allow easy creation of snippet that constructs a instance, and registering the snippet with a singleton responsible for providing a instance based on the string. Registration of the snippet is done in a constructor of specially crafted class instantiated as global variable.

Beware, builders implementation is both template and macro heavy.

See *generators*, *GeneratorFactoryCollectionModel::data*, *NewGeneratorWindow::buttonClicked* and *EditorSelectionWidget::provideEditor* for examples how to use builders.

[7]Classes `tree_node` and `tree_entity`

Storage

Storage subsystem lives in *util/storage.hpp* and is used in util subdirectories of gl and render directories. Useful for providing common objects like render models for box, coordinate system origin, etc. It is also used to load common shaders form cg subdirectory of program directory.

Bibliography

[1] Andersson J.: *Terrain Rendering in Frostbite Using Procedural Shader Splatting*, ACM SIGGRAPH, 2007

[2] Barrett S.: *Sparse Virtual Textures*, `http://silverspaceship.com/src/svt/`

[3] Mittring M.: *Advanced Virtual Texture Topics*, ACM SIGGRAPH, 2008

[4] Mei X., Decaudin P., Hu B.-G.: *Fast Hydraulic Erosion Simulation and Visualization on GPU*, Pacific Conference on Computer Graphics and Applications, Pacific Graphics 2007