Efficient procedural texture rendering for tile-based terrains

Eduard Zalyaev
DS2-2016
Innopolis University
Innopolis, Russia
e.zalyaev@innopolis.ru

***Abstract*—This document proposes fast pipeline for procedural tiled texture rendering. It is based on texture bombing technique. Proposed solution was tested against conventional C# tools that utilize delegates for Unity engine object updates.**

I. Introduction

Development of the terrain generation system for the course project required some kind of toolset for tile rendering. The original implementation was not scalable due to the enormous amount of costly sprite update calls that it generates. A more efficient solution had to be developed.

II. Original Setup

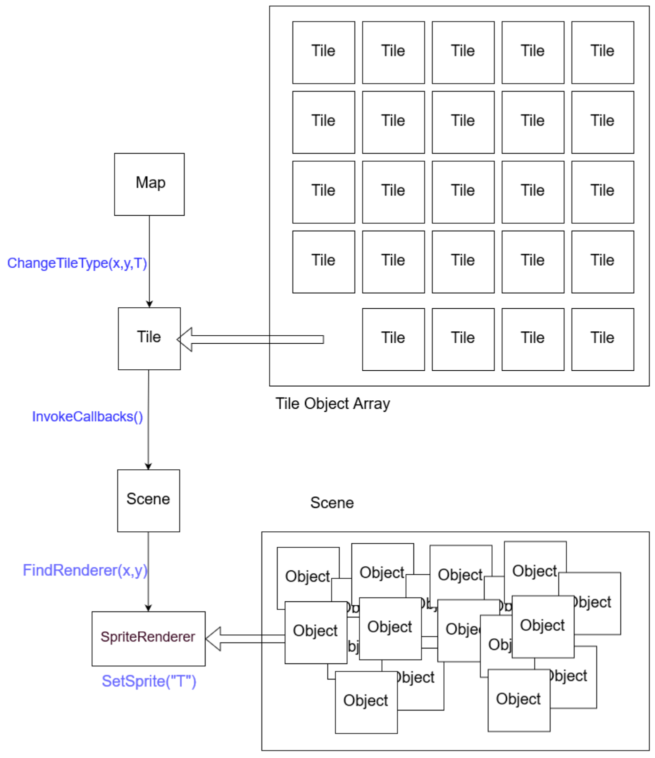

Fig. 1. Diagram representing a single tile update call

Here you can see the pipeline previously used for updating the tiles. The whole system is supposed to divide data and the representation of it into separate entities. Tile object classes contain a variable which represents the type and a few functions like isWalkable() or isWall() which are used by characters in order to traverse through the map. C# has a nice feature called delegates which can aggregate multiple function calls that belong to other objects. For example in the image above we have a model of the game world that for some reason wants to set a tile at coordinates x and y to be of a particular type T. It first tries changes the value of the type in the data model which in its turn calls all aggregated functions that are supposed to be called when a tile update is happening. One of these functions is supposed to find the sprite renderer in the scene which is responsible for drawing tile at position X,Y and sets it's sprite to the desired one. There are more routines that can be described further but they are not as important. There are some issues with this approach:

- **No batch updates**: we can't change multiple tiles at once.
- **Each update requires searching through the scene**: there are no references that would allow us to easily find sprite renderer for a particular tile.
- **All tile sprites are in memory**: the whole grid fills memory with useless duplicates of identical sprites. Imagine a chunk of 1024x1024 cells with 32x32 32bit images as cell sprites means that 4295MB of your VRAM is wasted. Add 1024x1024*6 vertices for each cell's plane to this and there will be no memory for you to use at all.

In order to procedurally generate terrain we would have to do that before the scene is displayed and controls are given to the player. That would add a few seconds to the loading screen and will stutter when some area have not been prebaked. Forget about realtime generation.

III. Texture Bombing

GPU Gems [1] book is a collection of tricks that can be achieved with shaders written for GPUs to offload simple computations. Chapter 20 of the book suggests a very nontrivial approach to procedural rendering of glyphs on a texture. Instead of manually drawing each repeating pattern on the texture image and sending it to the GPU why don't we just tell it where it is and tell it how it looks, how big it is. How do we do that? We can encode this information in another texture. Each pixel will represent a single element on the grid. Red and green channels can be represented as offsets relative to the cell position, blue channel can represent the scale of the element and alpha will tell GPU which image to pick from the set of available ones. This set is called sprite atlas. It contains all textures that have to be rendered by this particular encoding. The maximum amount of sprites that we can hold in this texture is 255 since the colors can only represent that many values.

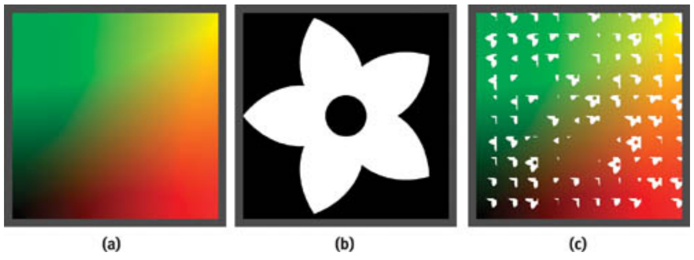

Fig. 2. (a)Original texture (b)Pattern that we want to put on the texture (c)Result of the texture bombing without neighbour consideration. Taken from [1]

As you can see from the Fig. 2 the flowers got clipped because each cell renders it's own flower. That is why we have to also consider neighbours of each cell, query their displacement and size and then render their pixels as well. There are details that would take up a lot of space to explain. Like how do we understand pixel's location on the grid or how do we make sure the pixels of overlapping elements do not saturate the result too much (this one can help understand how we can render transparent elements)? That is why reading the original text will be useful if needed.

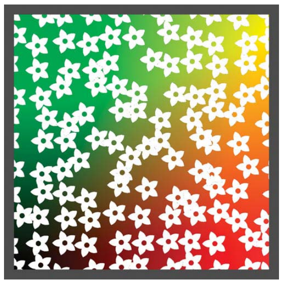

Fig. 3. Result of the neighbour aware texture bombing. Taken from [1]

IV. Implemented solution

Fig.4 illustrates current pipeline for a tile update. Notice that type change for rendering now requires only one operation of setting a pixel within the encoding texture. That allows us to change multiple tiles by batch updates of the pixels using Unity engine built in tools. And in order to do that we don't have to search for the object that is rendering the chunk. Required reference to that object is stored in the chunk object. More importantly one chunk of size 1024 requires 1024x1024 encoding map + 32x32xN tile atlas. That's less by a 1000 times less than the previous approach.

Additionally we can procedurally generate tile displacements. That way the terrain looks more natural because tiling can be barely seen.

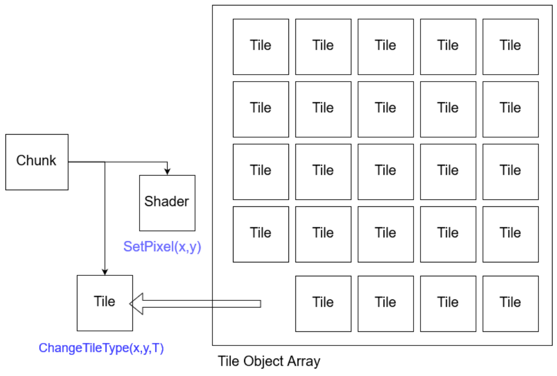

Fig. 4. Texture bombing based solution

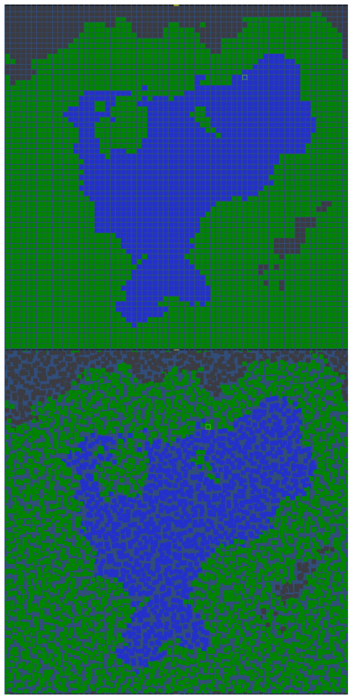

Fig. 5. Tiling with and without procedural offsets

V. Evaluation

Single 1024x1024 chuck of tiles creation and display was measured as well as single and 32x32 batch tile updates.

	Original	Texture Bombing
Creation	3m	15ms
Single update	0.81ms	0.43ms
32x32 batch update	56ms	16ms

VI. Conclusions

Implementation and assessment of procedural texturing technique showed promising results that will allow us to render large worlds at low cost and high speeds.

REFERENCES

[1] GPU Gems: Chapter 20. Texture Bombing *developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch20.html*