Texture Encoding For Dynamic Placement Of Background Detail Geometry

Johnmichael Quinlan

Abstract: In this report the author will describe the problems related to background detail geometry creation and level design. Specifically I'm going to concentrate on texture encoding methods to improve the usability and workflow for artists and designers while maintaining a solution in keeping with existing game engine technologies.

Index Terms: Texture Encoded Placement, Detail geometry placement, Level Design, Game Development, Content Editing, Content Creation

1 Introduction

In this report the author will describe the problems related to games development, specifically looking at the placement of large quantities of geometry that add aesthetic detail to the scene. The author will concentrate on how these collections of geometry are placed, edited and loaded. The author will explain the concept of detail geometry and will evaluate existing methods for editing the orientation of instances currently being used within the games industry. The author will propose a new data encoding and transport method for detail geometry instances. The author will show how the orientation and attributes of each instance are encoded and decoded, while using existing file formats and editing software already in use within the games industry.

1.1 Asset Pipeline Issues

Issues with content creation in the games industry focus on the following problems.

- Size of modern Levels
- Data transfer rates from disk
- Rendering cost of geometry
- Resource requirements of available hardware
- Scaling between available hardware platforms
- Available 3rd party software
- Custom bespoke software and its costs

The artist's ability to create realistic environmental assets comes second to the final requirements of the systems they will run on as described by Bethke and Ahern [1], 0. If a level is too expensive it will have to be modified in order to make it fit within a systems resource budget. Artists are constrained by the availability of tools to edit the game formats prescribed by the engine. The role of the tool chain is to mitigate this relationship by allowing the flow of data from an artist friendly digital content creation package into a game ready format. By doing this, the artist is freely able to wield their virtual brush and create masterful pieces of art, however the conversion of the artist's data to the engine data is not a perfect encoding. In most cases the conversion loses fidelity from the original data. Textures will be quantized, encoded and compressed giving the effect of reducing the textures fidelity. Vertex data will go through a similar process being tightly packed into data structures with limited precision as described by Aliaga 0.

2 Detail Geometry

Static instanced geometry is used to increase the realism of a scene without the memory cost of duplicated geometry per object. Its use is to increase the aesthetic feel of an environment as is seen with texture encoding methods for bump and detail mapping techniques 0, 0, 0. Some objects that lend them self's very well to the usage of detail geometry are:

- Trees
- Rocks
- Pebbles
- Grass
- Boxes
- Crates

3 Current Methods

3.1 Locator Placement

The positioning of a locator within a modeling application is a common place solution. The artist, while producing the underlying background geometry superstructure, can drop locators which reference a particular detail geometry type. They can set the orientation of the locator. This method can be achieved using Autodesk's Maya software package 0. The pipeline would perform a conversion step on the background asset to produce the data that the game engine will load. The conversion step is potentially increasingly expensive as the background geometry super structure gets larger. Meaning longer conversion times and the turn around on small tweaks makes the process unusable for designers and artists to test the level setup.

3.2 Bespoke Software

Bespoke software goes someway to solving this problem, generally allowing an artist to individually place detail geometry at a later stage in the pipeline. Effectively decoupling the background asset from the detail geometry data. This method allows artists to customize the positions without having to export from modeling applications upon each change. Viewing feedback from the placements is generally faster due to this. Ahern describes similar solutions [1]. Issues with rework need to be addressed by code that clamps instances to underlying terrain geometry. Updates to the terrain height are applied to the detail geometry upon load. This means extra runtime code for updating the vertical axis value during load time. Bespoke solutions need to be created and the cost of such may be undesirable depending on the resources available, O'Neill describes these situations in regards to navigation meshes that conform to background geometry topology 0.

3.3 Third Party Software

Applications such as Speed Tree and World Machine allow detail geometry production and placement. However the

placements are produced with volumes and are randomly placed within the volume by code generation 0. Editing of the detail geometry attributes is limited on a per detail geometry type producing very similar looking blankets of detail geometry throughout a scene 0. As with bespoke software, placements would have to be regenerated if the underlying terrains are modified or code put in place to clamp at runtime as discussed in section 3.2. Third party software can cost a lot to license and integrate into a pipeline as described by Ahern and Bethke [1], 0. However it is common place within game content pipelines as source asset generating packages. These include Maya, Photoshop, and Blender. It's generally advantageous to reuse the development already put into 3rd party software to reduce the overhead on the development team 0, [9].

3.4 Method Conclusions

Locators placed in the source asset are not suitable due to conversion times. Bespoke software solutions are expensive to implement and require resources that are costly to a developer. Third party and bespoke software both have to solve the invalidated work issue by either a runtime clamping method or by manual labor. Third party software provides a well known usable solution for artists and designers without the cost of bespoke software. Third party software already being used produces known formats that can be loaded in a game engine.

4 Proposed Solution

Using readily available software the author will provide a solution that can have the benefits of runtime geometry instancing using existing well known methods for data transport formats and methods. The author will aim to reduce the rework for artists by implementing existing solutions for clamping placements to underlying terrain geometry. The author will attempt to keep the solution below the cost of existing data transport formats both in memory and processing. The proposed data transport method is broken into two data stores. The first being a description of the individual detail geometry types. The second part being the attributes of the locator, including but not limited to, the locator orientation and type. The proposed file format for the description database is an XML file. The proposed file format for the locator encoding is a texture in the R8G8B8 format. These formats were chosen for their existing use within the games industry for loading of other types of data as seen in Aliaga, Sloan and Houska's work 0,0,0,0.

4.1 Texture Encoding

The texture encoding method chosen gives enough data to store or extrapolate the positional, rotational and scale elements of an instances orientation. The method also allows enough storage to encode the geometry type into the format.

Positional XZ Coordinates = Texture UV coordinates
Positional Y Coordinate = Surface Y coordinate
Rotational XZ = Surface normal of the location
Uniform Scale = Green channel of the texture.
Geometry Type = Red channel of the Texture
Geometry Y Rotation = Blue Channel of the texture

Software chosen to produce the encoded data was Adobe Photoshop in keeping with already familiar texture editing packages. Geometry types were setup in a custom application for creating the XML database.

4.2 Texture Decoding

To decode the data from the texture the author used the following method. For each pixel we calculate the index into the texture data array. The index is then used to extract the pixel data from the array. The following code snippet gives us our 3 data channels red, green and blue.

```
//creates an index into the array
int index = x * m_iStep + y;
//get the pixel from the array
D3DXCOLOR pixel = D3DXCOLOR(m_pPixels[index]);
```

Using the red channels data from each pixel the look up into the detail geometry database, loaded from the XML file, to retrieve the associated object for the red channels value.

```
//look up each pixels colour in the map
CGameObject * anObject =
(CGameObject*)m_colourKeyMap[pixel.r*255.0f];
```

With the object type extracted from the pixel we can extract the other channels data and apply them to the instance of the object. The position is already extrapolated from the pixels UV coordinate, although the Y value for the position is looked up in a cache of the heights from the height map landscape. This reduces the rework required if the underlying terrain changes.

```
D3DXVECTOR3 pos =
D3DXVECTOR3((float)x,m_pHeights[index],(float)y);
```

To decode the scale and rotational values from the pixel's data the author used the following code to set the instances orientation matrix.

```
anObject->SetIdentity();
anObject->SetScale(pixel.g*25.5f);
anObject->SetRotation(0.0f, pixel.b*6.28318531f, 0.0f);
anObject->SetPosition(pos);
```

5 Conclusion

The texture encoding method is a valid transport mechanism for editing and loading that data into a game engine. Its use can been seen across the games industry in many different techniques of rendering as described in section 3. However, rendering is the first example of how this method is not entirely successful. In the authors implementation the rendering was done using a fixed function pipeline. Data had to be decoded on the CPU at load time then passed back to the GPU as a matrix for the rendering of each instance of the detail geometry. There is a potential for improvement here by researching geometry shader approaches it might be possible to push the decoding step on to the GPU which would remove part of the CPU bottleneck. Textures are often used for additional data because they map onto objects using the UV coordinate system explicitly dictated by the artists 0. This provides a very good method of texture size control in regards to physical memory cost. UVs are generally packed into the smallest texture required to fulfill the quality requirements of the geometry. The texture encoding method unfortunately does not have the same hand crafted memory management.

Textures, at least in the author's implementation, were a blanket fixed size and covering a fixed terrain area. This leads to actual pixel usage being quite low in cases where only a few objects are being placed. The placement carries a fixed memory and CPU decoding cost even if there was no geometry going to be instanced by that pixel. For future implementations the author recommends that layers of texture be used and layered with custom resolutions to provide coverage where required at a resolution that provides the minimal memory footprint for the geometry placements. Such layered texture systems exist already in terrain system such as the Mega Texture approach pioneered by John Carmack 0. It is the conclusion of the author that a much greater level of control and flexibility could be achieved for background artwork using a custom file format not based on a texture. This would provide tightly packed data with a fixed cost data structure. Although that would not benefit from the stated GPU optimizations. The author would however recommend the use of the technique in this paper for large areas of detail geometry placements such as field of grass or hairs on a characters skin where a texture would map neatly to the target geometry and provide a conveniently usable solution.

ACKNOWLEDGMENT

The author would like to thank Dr Martin Hanneghan from Liverpool John Moores University for your guidance and direction in preparation of the author's dissertation from which this paper is an extract. The author would like to thank his wife for her patience and editorial input to the paper.

REFERENCES

[1] Ahearn, Luke. 2002 Awesome game creation, no programming required. 2nd Edition. Charles River Media. Massachusetts.

[2] Bethke, E. 2003 Game development and production. Wordware, Plano Texas.

[3] Ahearn, Luke. 2006, 3D game textures, Create professional game art using Photoshop. Elsevier Focal Press. Amsterdam.

[4] Daniel G. Aliaga and Anselmo A. Lastra. 1998. Smooth transitions in texture-based simplification. Computers & Graphics, 22(1):71--81, February 1998. ISSN 0097-8493. http://citeseer.ist.psu.edu/aliaga98smooth.html

[5] Various, 2003. Texture Space Bump Mapping. Game Programming Gems 1. Pages 191 – 193. Charles River Media, Massachusetts.

[6] Sloan, P. 2005. Normal Mapping for Pre-computed Radiance Transfer. Available Online: http://www.ppsloan.org/publications/NMPRT.pdf (02/03/2007)

[7] Houska, P. 2006 Directional Light maps. Available Online: http://stud3.tuwien.ac.at/~e9907459/directionalLightmaps.pdf (22/02/2007)

[8] Speed Tree, 2006 Foliage Creation for games.. Available Online: http://www.speedtree.com/

[9] Maya, Alias, 2006 Modeling Asset Production Software. Available Online: http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7635018

[10] World Machine 2007 Creation of realistic landscapes. Available Online: http://www.world-machine.com/

[11] O'Neill, J. C. 2004. Efficient Navigation Mesh Implementation. Journal of Game Development , Volume 1, Issue 1, Article 4. Charles River Media, Massachusetts

[12] Carmack, J. 2009. From Texture Virtualization to massive Parallelism. Available online: http://mrl.cs.vsb.cz/people/gaura/agu/05-JP_id_Tech_5_Challenges.pdf