SIMULATING INFINITE CURVED SPACES USING VERTEX SHADERS

M. C. Bouterse,
A. Eliëns,
Department of Computer Science
Faculty of Sciences, Vrije Universiteit
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
E-mail: eliens@cs.vu.nl

KEYWORDS
Shader Programming, 3D In-Game Animation, Vertex Manipulation, Curve Simulation

ABSTRACT

Rendering dynamically curved meshes can be complicated if not impossible using traditional methods. Either it involves creating pre-curved meshes and fixed animation sets or processor intensive calculations to deform mesh data on the fly. A model is presented here that can be used to create infinite dynamic curved spaces in real-time using a vertex shader. This makes dealing with curved meshes in real-time rendering easier and makes procedural animations of curves possible. Applications include dynamically curving environments and objects for games.

INTRODUCTION

Shader programming has acquired an important position in the field of real-time rendering. The ability to control the rendering process by executing custom programs has provided new possibilities for numerous applications. Many articles have been published in the *ShaderX* and *GPU Gems* book series describing new algorithms for graphics hardware. Most of these articles demonstrate techniques to improve the visual quality of the rendering process. The ability of the vertex shader to manipulate vertices to animate or deform meshes is less often explored. A few articles dealing with this topic use the vertex shader to render ocean water (Isidoro 2002a and Finch 2004), fields of grass (Isodoro 2002b and Pelzer 2004) or soap bubbles (Isodoro 2002c). These articles show the potential of the vertex shader in vertex-based animation and deformation for very specific applications. This paper builds on the concepts presented in these articles and describes a general technique for applying curves to vertex spaces.

This paper presents a method for using vertex shaders to create infinite curved spaces; to apply curves in real-time to meshes. The algorithm provides an easy way to apply multiple curves in any direction to an arbitrary vertex space. First a straightforward method for creating a single curve is explained. This method is then used as a starting point for creating a more generic model. Finally our conclusions regarding this model are presented.

APPLICATIONS

The method described here is originally developed to create curved tunnel systems for a yet unpublished game. The method proved to be very useful for creating dynamically curved environments and can also be used for procedurally applying curves to in-game objects. The method allows for smooth curve animations that are hard to create with traditional technology.

APPLYING A SINGLE CURVE

To be able to apply a curve to an arbitrary vertex space, we need a per-vertex algorithm that translates each vertex from the original vertex space to the desired position in the curved space. For a single curve this algorithm is relatively straightforward to find. To illustrate the concepts presented here we use a cylindrical mesh centred on the positive z-axis starting from the origin. A cylinder is used here for illustrative purposes, the algorithm works on any vertex space, no matter what the distribution is. A cylinder is appropriate, because one of the most obvious applications, creating curved tunnel systems, uses meshes that resemble cylinders.

In figure 1 the result of applying a ninety degrees curve to the sample vertex space is shown. The length of the segment is denoted by the letter `d`.

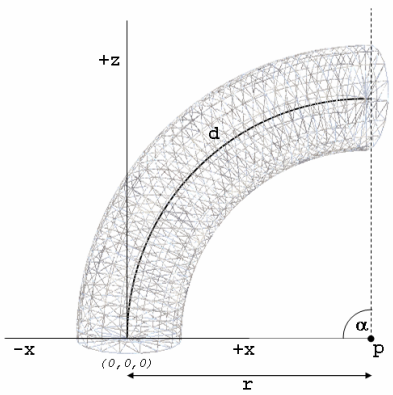

Figure 1: Applying a 90° Curve to a Cylinder

The length of `d` is preserved in the middle of the cylinder. Preserving the length in the middle of the vertex space is desirable in most cases and leads to minimal stretching and

squeezing of the original mesh. The length of the curved space will be the same as that of the original space.

Calculating this curve can be done on a per-vertex basis. The length of the segment (d) and the angle of the total curve (α) must be known beforehand. From this information the main radius (r) of the curve can be calculated using the formula for the circumference ($2\pi r$). Taking into account the part of a full circle that the curve covers, the formula for the radius becomes:

$$r = d / \alpha \text{ (with } \alpha \text{ in radians)}$$

This radius is the distance from the pivot point (p) to the centre of the vertex space (z-axis). This value is not used directly, but used to calculate the per vertex radius as will be shown in the next section.

To apply the 90° curve per-vertex the z-value of the position is used in the vertex shader to determine the position within the curve. For each vertex new coordinates are calculated. Because the curve is applied in the xz-plane, the y-coordinate of the vertex will be preserved. The x and z coordinates can be computed using basic trigonometry with the following formulas:

$$x' = x + r' - \cos(\beta) * r'$$
$$z' = \sin(\beta) * r'$$

Where r' is the per-vertex radius (r' = r − x) and β is the per-vertex curve angle that depends on the z position of the current vertex (β = (z/d)*α); the further away from the origin, the more the vertex will be displaced.

These are the calculations needed to apply a curve to a single segment with a certain angle in the xz-plane. To implement this in a vertex shader the following code snippet can be used:

```
// Compute per-vertex angle
float beta = (pos.z / d) * alpha;

// Compute per-vertex radius
float radius = r - pos.x;

// Calculate curved positions
pos.x += radius - cos(beta) * radius;
pos.z  = sin(beta) * radius;
```

The resulting shader is capable of applying a curve to a segment of arbitrary length by a variable angle. Although this might have some value in practise, the model is very restrictive; only one segment, starting at the origin can be curved and the curve is always in the same direction (in the xz-plane and towards the positive x-axis). A generic model for creating infinite curved spaces is needed and will be presented next.

GENERIC CURVE MODEL

To overcome most of the restrictions of the single curve model, we present a generic model that has the following additions:

- Curves in any direction
- Multiple curves that seamlessly connect

The algorithm we have so far will be used as a basis for creating such a generic model.
First the curve algorithm will be extended to support any curve direction. A naive approach is to simply rotate the model along the z-axis after the curve has been applied to turn it into the desired direction. This allows for curves in any direction, but introduces the problem that vertices are not in the same position as they would be if a curve was directly applied in that direction. A solution to this problem is to first rotate the vertices over the z-axis in the opposite of the desired direction, then apply the curve as shown before and finally rotating it over the z-axis again to its final position. The curve algorithm is now extended with a new parameter, a rotation angle over the z-axis. From now on we will use γ to denote this angle.
The second addition in the generic model is the support of multiple curves. Eventually the following algorithm needs to be executed for each vertex:

1. Determine the curve this vertex belongs to
2. Translate start of the curve to the origin
3. Rotate around z-axis by -γ
4. Apply the single curve algorithm
5. Rotate around z-axis by γ
6. Align with end of previous curve
7. Connect to previous curve

We have already shown how to implement steps 3-5; the remaining steps are needed to support more than one curve. Before we can implement the remaining steps we must divide the vertex space in segments (each segment has separate curve parameters). The first segment starts at the origin and ends at the plane $z = d_1$, the second starts at $z = d_1$ and ends at $z = (d_1+d_2)$, etc. Each of these segments has its own values for α, γ, and d.

The first step of the algorithm can now be implemented by comparing the z value of the current vertex to the start of each segment until the right segment has been found. Translating the segment to the origin is done by subtracting the start value of the segment from the z value of the current vertex. After the translation, steps 3 to 5 are applied as described before. This can be implemented by a single matrix multiplication. This matrix is the result of concatenating a rotation matrix (-γ) with a curve matrix (described next) and finally another rotation matrix (γ). The curve matrix needed here can be created from the calculations shown for the single curve algorithm. By substituting the radius equation (r' = r − x) into the

curve calculations, we get the following matrix multiplication:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \begin{bmatrix} \cos(\beta) & 0 & -\sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ (1-\cos(\beta)) \cdot r & 0 & \sin(\beta) \cdot r & 1 \end{bmatrix}$$

Finally the segment must be translated and rotated in such a way that it seamlessly connects to the previous segment (which can have an arbitrary curve as well). A transformation matrix must be computed that takes care of this. Given the per-segment parameters (γ, α and d), such a transformation matrix can be computed for each separate segment. The transformation matrix for segment N can then be computed by concatenating the matrices from segment `0, 1, …, (N-1)`.

The transformation matrix for each segment is computed by first orienting the vertices in the right direction and consequently translating them to the end of the previous segment. Aligning segment `N` with the end of segment `N-1` can be done by rotating the segment using the curve parameters of segment `N-1`. First segment `N` is rotated along the z-axis with the negated γ value of segment `N-1`, then it is rotated along the y-axis with the α value of `N-1` and finally it is rotated along the z-axis again using γ from `N-1`. The final step is to connect segment `N` to `N-1`, which can be done with a translation vector from the origin to the centre of the end of segment `N-1`. Using basic trigonometry we can calculate the components of this vector T:

$$T_{xy} = r - r * \cos(\alpha)$$
$$T_x = \cos(\gamma) * T_{xy}$$
$$T_y = \sin(\gamma) * T_{xy}$$
$$T_z = \sin(\alpha) * r$$

This is the final piece of the algorithm for a generic curve shader. Implementing this algorithm in HLSL or another shader language should be straightforward using the presented calculations. The per-segment parameters and the segment offset matrices need to be supplied as shader constants once per frame.

CONCLUSIONS

In this paper a method was presented for creating infinite curved spaces on the GPU that can be used to dynamically apply curves to meshes. In a few steps a flexible model was presented that is capable of applying multiple curves in any direction. The scalability of the implementation is depending on the available shader constants, which is probably only an issue on targets lower than shader model 3.0. Applications in game development consist of creating curved tunnel systems and other curved objects without the need for more assets or complicated animations. The presented technique works best on fairly dense and equally distributed vertex spaces. Meshes with few polygons will not look good when curved because the algorithm doesn't add vertices and curved objects need relatively many vertices. This method might be used as a basis for creating interesting dynamically curving game objects and environments. Further research is needed to determine the full potential and scalability of the presented technique. The per-vertex calculations are relatively heavy, but because most modern games are not bottlenecked by vertex processing capabilities, this method should not lead to dramatic loss of performance.

FUTURE POSSIBILITIES

The presented algorithm and implementation runs sufficiently efficient on SM 3.0 hardware, but is still limited by the amount of vertex constants available. This limit will be almost gone on DirectX 10 hardware. Shader model 4.0 supports considerably more constants, making this limit much less important. Also the geometry shader introduced by SM 4.0 could lead to interesting new applications of the algorithm such as the generation of curved objects on the fly or generating extra vertices to curve meshes with few polygons smoothly.

BIBLIOGRAPHY

Isidoro, J.; A. Vlachos, C. Brennan. 2002a. "Rendering Ocean Water" In *Direct3D ShaderX*, edited by W.F. Engel. Wordware Publishing.

Isidoro, J. and D. Card. 2002b. "Animated Grass with Pixel and Vertex Shaders" In *Direct3D ShaderX*, edited by W.F. Engel. Wordware Publishing.

Isidoro J. and D. Gosselin. 2002c. "Bubble Shader" In *Direct3D ShaderX*, edited by W.F. Engel. Wordware Publishing.

Finch, M. 2004. "Effective Water Simulation from Physical Models" In *GPU Gems*, edited by R. Fernando. Addison Wesley.

Pelzer, K. 2004. "Rendering Countless Blades of Waving Grass" In *GPU Gems*, edited by R. Fernando. Addison Wesley.

AUTHOR BIOGRAPHY

ANTON ELIENS studied art, psychology, philosophy, and computer science. He is lecturer at the Vrije Universiteit Amsterdam, where he teaches multimedia courses. He is also coordinator of the Master Multimedia for Computer Science. He has written books on distributed logic programming and object oriented software engineering.

MARCO BOUTERSE received a Master degree in Computer Science/Multimedia at the Vrije Universiteit Amsterdam. His master thesis was about the development of games with a focus on shader technology. Currently he is working as a game programmer at Two Tribes, a game company located in Harderwijk, The Netherlands.