Innovations Report: Procedural Fur System

Joe Withers*
National Centre for Computer Animation

Abstract

For this project I developed a procedural fur system, with the aim of exploring the feasibility of offloading computation onto the GPU within artist tools. The final artefact is an application which serves to interface with the fur system API that I have developed. This report primarily documents the implementation of the API, as well as my findings.

1 Introduction

Introduction Text.

1.1 Existing Solutions

Existing Solutions. (Xgen, houdini)

2 Implementation

Method Text.

2.1 Resources

I decided to develop the API side of the fur system using C++, primarily as it most commonly used when developing computationally heavy artist tools, but also because it is the language I am most comfortable using. I opted to use OpenGL over other APIs (CUDA, OpenCL) for offloading of computation onto the GPU, simply because my final artefact is a tool with a graphical interface, and OpenGL is capable of handling both arbitary computation using compute shaders and rendering of geometry simultaneously.

To handle the user interface I used the Qt framework within C++. Qt is commonly used for artist tools within visual effects as it is cross-platform, and applications can be configured to run within other applications that make use of it, such as Autodesk Maya.

I wanted to include a node-graph style interface within my application, as it is commonly used within existing artist tools (Autodesk Maya, Unreal Engine), and would encourage modularity within my API. Qt does not natively provide this kind of interface, so I made use of NodeEditor [Pinaev 2017], an existing Qt-based library that provides this functionality.

2.2 Design

From looking at existing fur systems I was able determine that my simplified fur system would need to consist of the following components:

- Geometry Loaders - These are responsible for loading user specified geometry. Example Geometry Loaders could allow for the parsing of Wavefront OBJ files.
- Distributors - These are responsible for the distribution of curves onto user specified geometry. User controllable parameters could include density (curve count), distribution pattern (random, uniform), and curve length. These parameters could potentially be controlled by texture inputs.
- Operators - These are responsible for manipulation of the curves to achieve the desired look. Example operators could provide bending, clumping, or randomisation of input curves.
- Renderers - These are responsible for the rendering of curves into the application viewport. Example renderers could provide mesh, cuves as lines, or curves as ribbons rendering functionality. User controllable parameters could provide controls for the shading model in use, as well as control of the base and tip widths when rendering curve ribbons for example.

Whilst the list of components specified above provide the required functionality for a standalone tool, they do not provide any functionality for exporting fur for use in a broader pipeline. I chose to omit this functionality to limit the scope of this assignment, but the ability to export hair curves as alembic would be desirable, as alembic is a widely adopted format that allows for the storage of multiple different types of curve geometry. Another candidate file type would be RenderMan's RIB format, which would allow for direct rendering of the hair curves in any RenderMan compliant renderer.

2.3 Development

write about obj loading

write about distributing https://stackoverflow.com/questions/9294316/distribute-points-on-mesh-according-to-density

write about GPU

3 Results

Results Text.

3.1 Performance

insert stats comparing CPU/GPU compute

References

PINAEV, D., 2017. Qt5 node editor. `https://github.com/paceholder/nodeeditor`. Accessed 25 Jan 2018.

*e-mail:joewithers96@gmail.com