Editable Toon Shading Tool in Maya

Masters Project Thesis

Yiyang Huang

S5303439

MSc Computer Animation and Visual Effects

Abstract

Toon shading is widely used in anime and game development. However, only using procedural shader method could not always produce desirable shading effects. Artists probably need lighting or shading effect for specific areas on mesh surface, which is required a tool to make it work. In this thesis, the development of an editable shading tool is explored in Maya, trying to make shading effect controllable for artists.

Contents

1. Introduction

Toon shading, also known as cel shading, is a general artistic expression used in animation and game productions. Compared with realistic rendering, it does not pursue to simulate the illumination in real-world but keep the art style of hand-drawn cartoon instead.

However, this procedural process still has its disadvantages in animation and game development. All the shading effects are generated based on the interaction between lighting and geometry data. Sometimes artists could obtain some desirable shading effects especially it is hard to adjust the surface normal, or sometimes artists might need more shading area but are restricted by the geometry itself. Although some methods could help adding extra shading manually as post-process, they are still unavailable for real-time rendering.

A tool could be developed to meet the demands of making toon shading controllable. Artists could add any control point or locator on the rendered surface to modify the rendering details, making the area to be lit or shadow. In order to obtain the desired shading details, each edit has some parameters to adjust its shading shape. In addition, every shading detail modified by this tool should be dynamic, which means these shading areas could be adapted to the dynamic light changing instead of being fixed on the mesh surface. Users could also interpolate keyframes to these edit points or locators for a desirable effect.

2. Previous Work

The demand for controllable light and shade on stylised shading has already existed in animation and game industries. For toon shading, the shaded area is calculated by the normal and light direction. Therefore, obtaining a desirable

shading in specific area requires to modify the geometry or its normal map. This solution is not always flexible because it would result in an undesired shading area again when light direction changes or object moves. Moreover, the costs of altering geometry are too great, especially when the model could not be reused for other occasions. Thus, the prevalent way is to add shaded area manually during post-process, which is time consuming.

Manual shading is feasible in animation production process but not available for real-time rendering like game runtime. In order to resolve this problem, some researches propose their own methods to design the shaded area and apply the keyframe interpolation.

2.1 Controllable Stylised Shading

The controllable stylised shading was firstly proposed by Todo et al. (2007) In his work, a painted-brush was used to design the shaded area, whose data were recorded in each vertex. His main research focuses on keyframe interpolation for the user-defined shaded area. As shown in Fig. 1, the offset which was calculated by user-defined shading threshold and dot production result of light direction and normal, was used to distribute new shaded area. The interpolation for the offset value used a radial basis function (RBF) based on research by Turk and O'Brien (1999). In addition, they also built "intensity brush and smoothing brush" to allow fine adjustment.

However, this method has its shortage. If two keyframes have long interval, the interpolation could become discontinuous, which means more keyframes are required to keep shaded area moving smoothly.

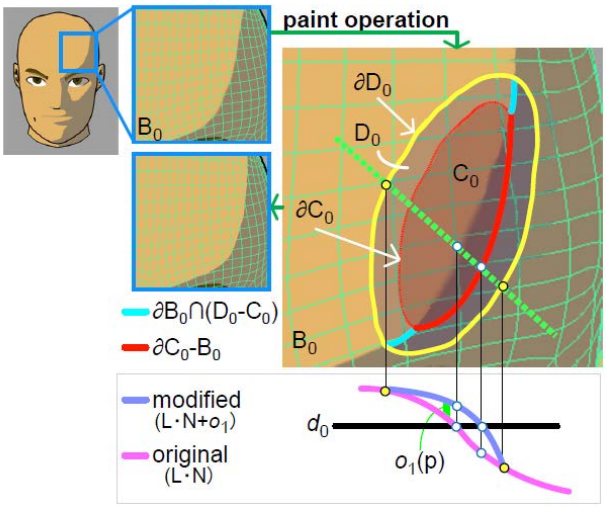


Fig. 1 Locally Controllable Stylised Shading (Todo et al. 2007)

2.2 Shading Rig

The newly proposed method, called Shading Rig (Petikam et al. 2021), improved keyframe interpolation problem mentioned above. The RBF used in his paper was spherical radial basis functions (Tsai and Shih 2006), which could preserve the continuity of the shaded area.

The way of user-defined lighting in his work used locators instead of painted-brush as Todo et al. (2007) did. It worked like light projection on object surface but defined its own texture space, which conducted the lighting area based on the warped texture coordination. To be artist friendly, each edit locator had various parameters that could control the shaded shape (seen as Fig. 2). The definition of the texture space referred to the dynamic lit-sphere (Todo et al. 2013) usage for stylised shading, which will be mentioned in Section 3.2.

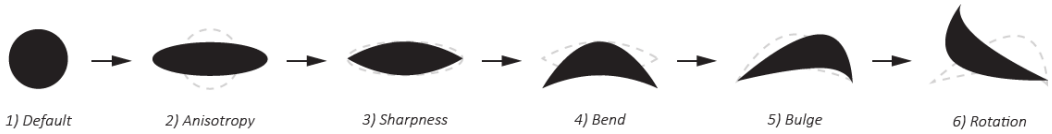


Fig. 2 Parameters for each edit to manipulate the shape (Petikam et al. 2021)

3. Technical Background

3.1 Cel Shading

Cel shading, also called as toon shading, was first described by Philippe Decaudin (1996). It is a type of non-photorealistic rendering (NPR) which uses few colours to make object looks artless. The colour of the cel shading is based on the different steps of light diffuse on the surface. For a two-step cel shading, the lighting part satisfies the following condition:

$$dot(L, N) > d_0$$

where *L* is the light direction and *N* is the surface normal. The shading threshold is defined as d_0, which decides whether this pixel should use light colour or shadow colour. For three or more step cel shading, the similar method can be applied.

3.2 Dynamic Lit-Sphere

Lit-sphere was originally designed to capture the feature of NPR shading. It can be used to generate stylised shading feature via a sphere map. The shading details are record in this sphere map and can be transferred into corresponding texture coordination based on the direction of the sphere normal. This method is also applied as one of the techniques of material capture (MatCap).

The lit-sphere model made by Sloan et al. (2001) was based on the viewport direction. This approach is efficient to capture the shading details from a static scene. However, it could lose its shading tones from an environment with dynamic light.

In order to obtain the better shading effect under dynamic lighting, the proposal

of dynamic lit-sphere (Todo et al. 2013) redefine the texture space transformation, which replaces the viewport direction with the light direction. The light space normal is defined as $n_l = (n_{lx}, n_{ly}, n_{lz}) = (n \cdot l_x, n \cdot l_y, n \cdot l_z)$, where *n* is surface normal and $l = (l_x, l_y, l_z)$ is light space defined by the light direction. Then final shading colour can be sampled by $(u, v) = (rcos\theta, rsin\theta)$, where $r = \arccos(n_{lz})/\pi$ and $\theta = \arctan(n_{ly}/n_{lx})$. The shading sample for each edit used below is based on this approach.

3.3 Maya Plugin and Viewport 2.0

The purpose of this project is to build an editable shading tool for artists to use. Maya is a prevalent choice for modelling, animation rigging and rendering. It also offers artists a good environment to preview their work. Thus, Maya was chosen to build a plugin for adjusting shading effects.

The Maya Python API 2.0 is a new version of the Maya Python API which provides a more Pythonic workflow and improved performance (Autodesk Maya 2020). Compared with its old version, the new version still preserves most of previous features and run faster. The biggest advantage of using Python for Maya plugin development is that it is easier to compile and debug code than using C++, though plugins built by C++ are more efficient than Python.

Maya 2012 introduced a new set of API classes for defining custom drawing, shading and effects in both Viewport 2.0 and Hardware Renderer 2.0 (Autodesk Maya 2020). Its brand-new rendering architecture has high-performance for large scene rendering. However, some development details for rendering part are not documented well, which makes confusing for developers.

4. Implementation

To make toon shading editable, a Maya plugin was built base on the Maya Python API 2.0. Python language was selected for its efficiency on plugin development. For the Maya version, the plugin was developed in Maya 2020 so the Python language version used was 2.70. The plugin was only tested on Linux and Windows systems and was compatible with both systems. No extra extension or library was used in this plugin. This plugin also includes an installation script (seen as Appendices), which allows plugin installed or uninstalled more efficiently.

The plugin was developed mainly based on the research work mentioned in Section 2.2. In the following sections, the basic code structure will be presented. Moreover, the usage of shading tool will be introduced as well as some development process being indicated.

4.1 System Structure

4.1.1 Overall Structure

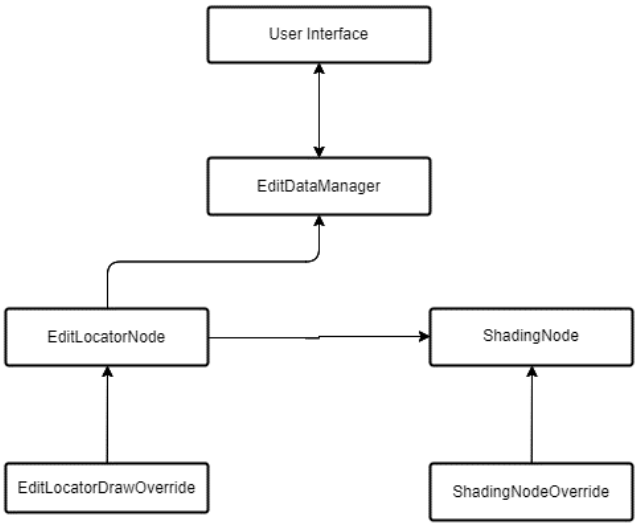


Fig. 3 Overall structure of plugin code

In general, the structure of this plugin system (as seen in Fig.3) could be divided into three parts: user interface, edit locators and internal shading, where user interface was built with Maya command line by Python and other two parts were built mainly based on Maya nodes.

The object presented on Outliner window are not always the object itself. When users select the object, they usually select the transform of that object as parent object. In order to organize all the edit locators in a better way, an edit data manager was built to record all the locator nodes and their parent transforms as well as the mesh object they belong to. The data manager allows the newly created nodes to be reused in an efficient way and not need to be retrieved again via other complex routines.

For each edit node, it was registered with a draw override, where a simple coordination graphic was drawn to display its location in Maya Viewport. Users

can easily figure out where the locator is placed and select it directly in Viewport. In addition, this draw override is a hardware rendering, which means its graphic drawing is conducted by GPU.

Shading node is where the major toon rendering takes place. This node was inherited from a basic dependency node and was implemented to be shown in Hypershade window. It was registered with a “ShadingNodeOverride” and turned into a hardware rendering node.

4.1.2 Node Connection

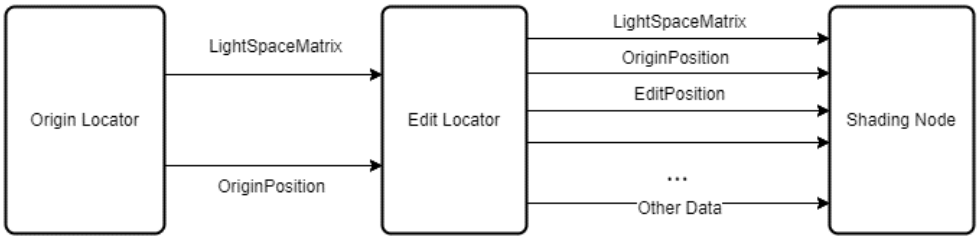


Fig. 4 Node Plug Connection

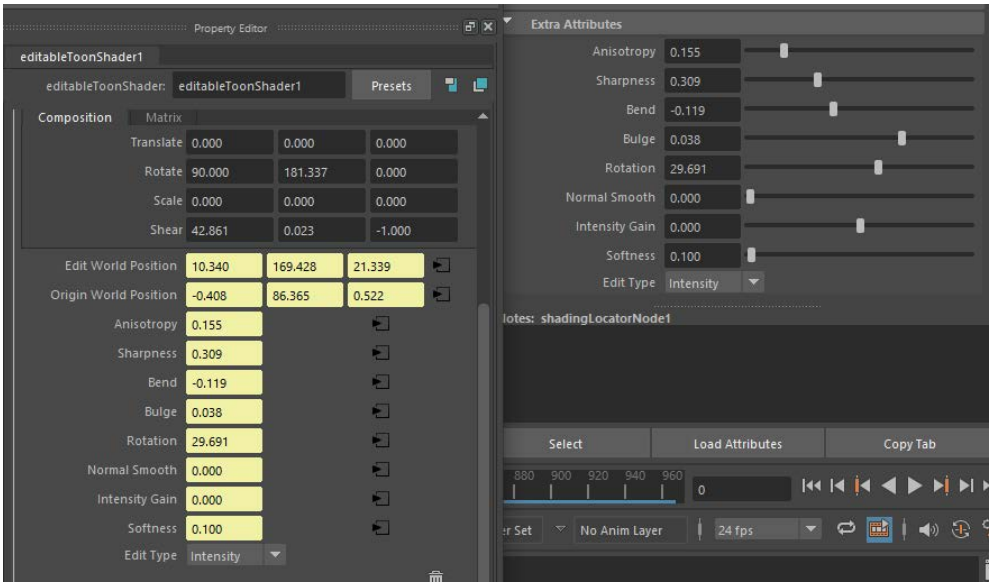
As shown in the initial design, the shaded area is shaped by the position and other parameters of the edit locator. Whenever the edit locator makes a change, the corresponding data should be reflected into the shading node in time. This requires a connection between two different nodes and update data in an immediate way.

In Maya, attributes can be connected by plugs. Once the data in an attribute have changed, the connected attribute will be set dirty and be evaluated afterwards. This mechanism ensures every data modification from the other node can be immediately received and the current node can update or respond other nodes in an effective way.

The major node connections in this plugin are illustrated as Fig. 4. The origin locator belong to the edit pair is used to defined the texture space for each edit, which will be mentioned in Section 4.3 later. Meanwhile, the corresponding light space matrix should be computed in the node instead of in the shader, because it is better to be used as a uniform variable in shader rather than being calculated for each vertex streamflow.

In shading node, a compound data attribute was set to connect with all the edit locators. Then all these data would be processed and be passed into shader.

A specific point needs to be noted is that the world position for each edit locator or its origin locator cannot be obtained from the node itself. In fact, all the position data can only be found in its parent transform. Therefore, the connection source is actually the parent transform of source node instead of source node itself.



4.2 Interactive Tools

4.2.1 Assign Toon Shader

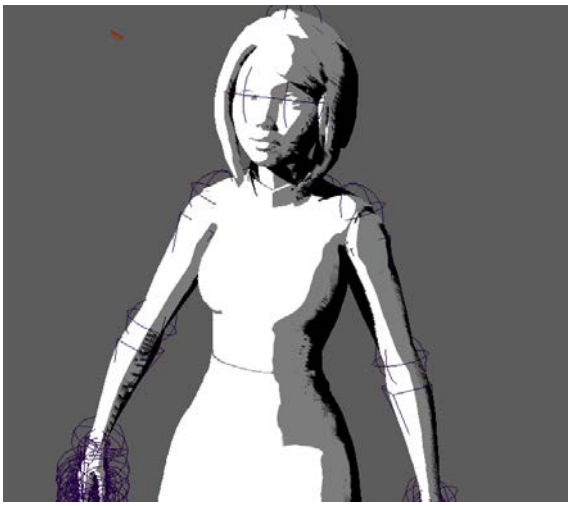


Fig. 5 Assign toon shader on mesh surface

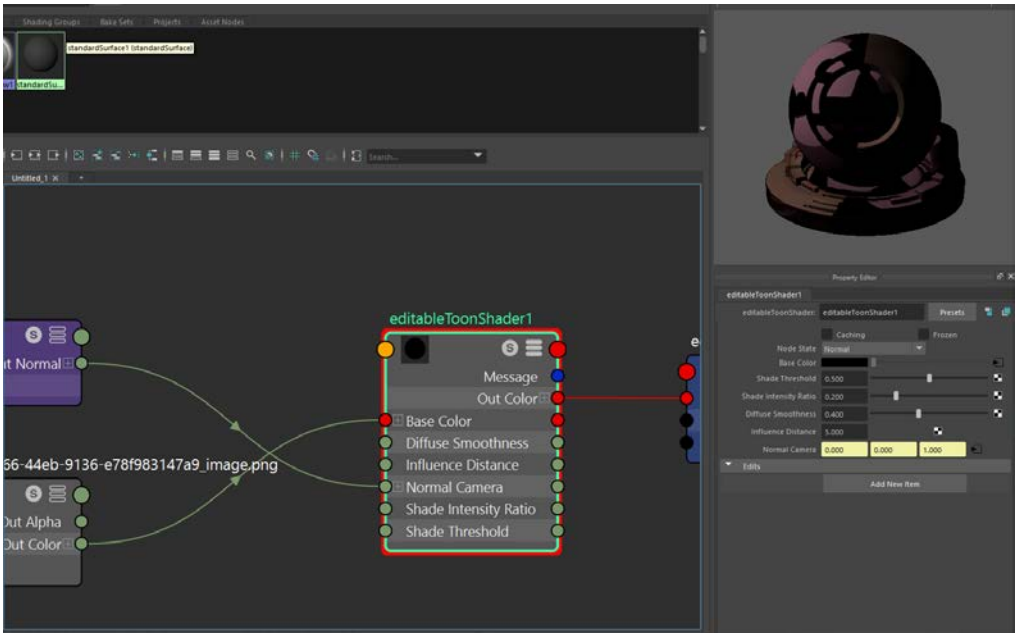


Fig. 6 Connect shading node with colour map and normal map

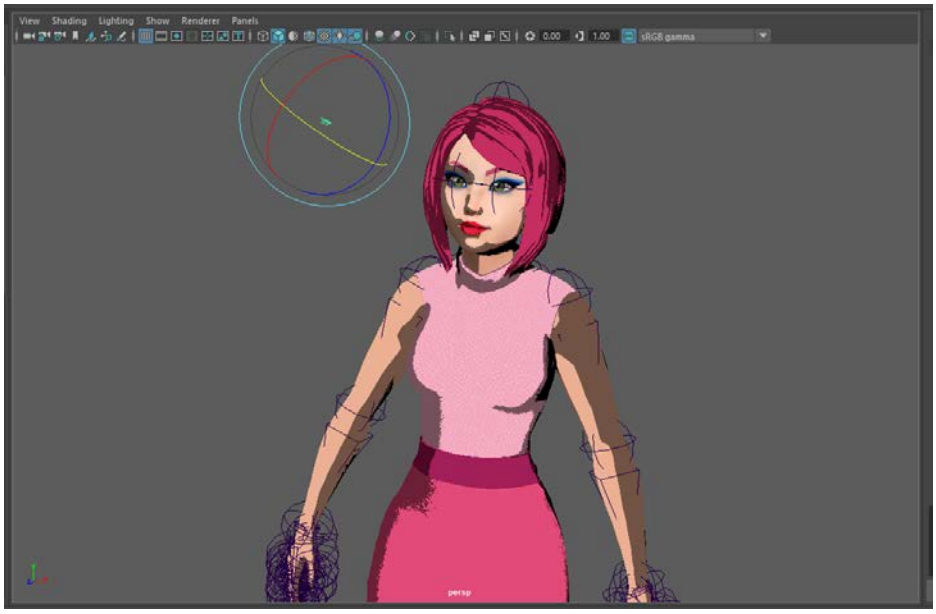


Fig. 7 Toon shading after colour map and normal map applied

The toon shader existing in Maya cannot be employed as the task shader. In order to accept data from edit locators, a new toon shader was built. This shader is a basic 2-step toon shading process and the shading node is accessible for normal map. The implementation of normal map used default fragment built in Maya.

4.2.2 Create Edit Locator

After the custom toon shader was assigned, edit locator could be added to control the shading area. By default, the locator will be added at world origin if not specified. This could be troublesome to find the locator if the model mesh is too big or camera does not face to the origin of the world coordination.

The tool designed here has already solved this problem. A new edit locator would be created at a proper position based on the viewport (camera view) facing currently. This could save time for users to roll the screen in order to find this edit locator.

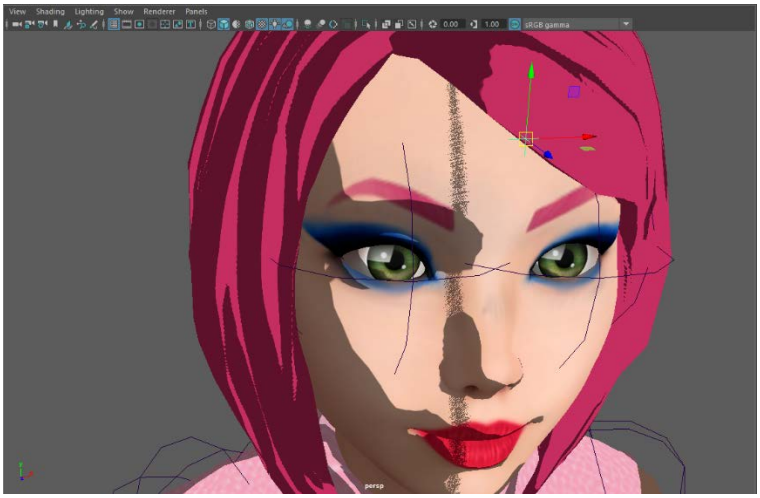


Fig. 8 An edit locator will be automatically created based on current viewport position

Maya API provides a method to get the current viewport position in world space. Meanwhile, there is a mesh function which can obtain the closest vertex position on the mesh by the specified world position. The final world position for edit locator to be created could define as following:

$$p_l = lerp(p_m, p_v, k)$$

where p_m is the closest position of mesh vertex and p_v is the world position of the viewport centre. *k* is the interpolation value, which is set to 0.1 by default.

4.2.3 Move Origin Locator

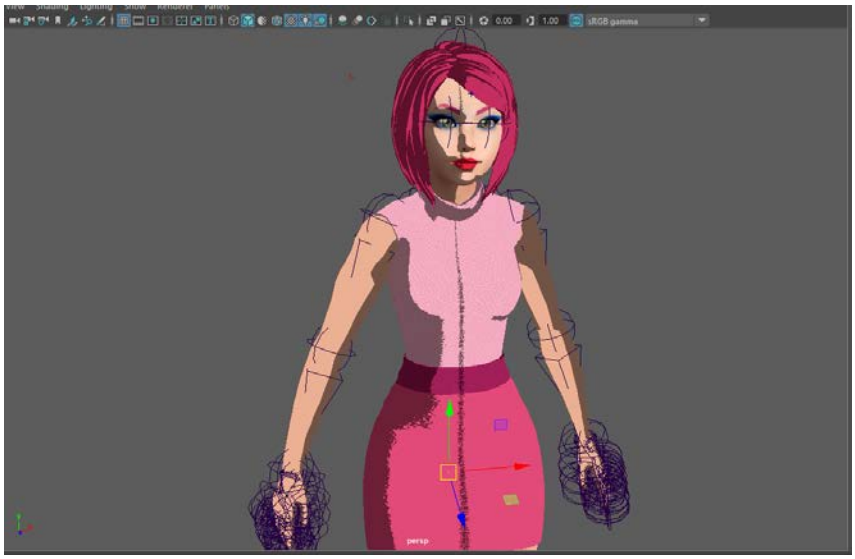


Fig. 9 Locate to the corresponding origin locator

The origin locator also takes part in the shading controlling. By default, it is set to mesh centre when created. To obtain a desired shading, users still need to adjust its location.

Similar to creating and obtaining an edit locator, the tool is designed to find the corresponding origin locator to the current selected edit locator. This process is fast completed via retrieving the data manager mentioned in Section 4.1.1.

4.3 Shading Edit Locator

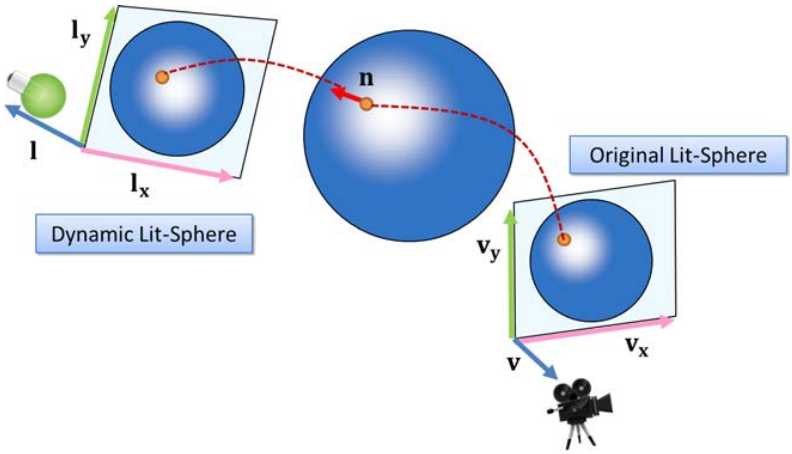


Fig. 10 Dynamic lit-sphere used to capture stylised shading details in dynamic light environment (Todo et al. 2013)

In this task, shading is mainly controlled by the edit locators. Origin locator in this task mainly works as a pivot to generate texture space for shading projection.

As mentioned above, the texture space is generated via the approach of the dynamic lit-sphere. The only difference is that no surface normal takes part in this calculation. The normalized direction from world space position of vertex p_w to world space position of edit locator p_l is used to replace the surface normal.

As shown in Fig.2, Bend (wy), Bulge (wx), and Rotation (θr) are used to inflect and rotate the shading shape. Defining w = (wx,wy)⊤ and u = (u, v)⊤, a warping function θw(u) is shown as following:

$$\theta_w(u) = k_w \cdot (R(\theta_r)u)$$

where kw is set to 10. R(θ) is the 2D rotation matrix given an angle θ. Then the warpped (uw, vw) coordination could be computed as:

$$(u_w, v_w) = w + R(\theta_w(u))(R(\theta_r)u - w)$$

With the modification for warpped texture coordination, the intensity of edit lighting could be sampled on it. Then edit lighting intensity is defined as:

$$I(u_w, v_w) = e^{-au_w - \frac{1}{a}|v_w|^{2-s}}$$

where a is Anisotropy and s is sharpness.

Based on this intensity distribution, all the edit lightings can be accumulated to generate shading area.

4.4 Shading Node

4.4.1 Hardware Rendering

As mentioned above, a “ShadingNodeOverride” was registered with the dependency node to implement hardware shading. The rendering work in Maya Viewport 2.0 is organised by an XML format called as fragment. The real shader here is call as effect, which can be linked or writing raw text directly in an XML fragment. Then all the fragments are connected together by a fragment graph, which will be compiled to form the final rendering outcome.

The basic toon shading has already been implemented for all display drivers. However, some technical details about fragment implementation are not mentioned, which leaves a big hole to do the black box test. The problem here is how to pass all the edit data into shader and obtain the desired shading effect.

4.4.2 Solution Attempts on XML Fragments

Two solutions are tried in shading node so far. Only the last method is feasible somehow but still has many constraints.

The first solution is to regard all the shading edits as lights. The effects of these shading edits work like multiple lights on the surface rendering in a specific aspect. Thus, imitating the rendering process of lighting could possibly resolve the problem. As presented in XML schema (Autodesk Maya 2020) and examples of development kits, two fragment types, “selector” and “accum”, could possibly be used for rendering by multiple shading edits. However, all the examples shown by using this method all take advantage of the default lighting fragment built in Maya. Therefore, it cannot be confirmed this method is correct.

According to the example of phong1 shader in XML schema, the fragment graph still needs a struct type as input. Considering the same storage way for struct and array in memory, an array data was attempted to pass into the fragment graph. But it was not accepted by the fragment graph even though the fragment graph with custom ‘accum’ and ‘selector’ fragments was compiled successfully.

The final solution is to pass array data directly to fragment shader. However, this approach is only available for Cg implementation. For HLSL and GLSL implementation, the shader effects would fail to be compiled since the array parameters were defined. Because this situation still happened even if parameters did not take part in calculation or no array data were passed in shader. The assumption is these two types of shaders cannot accept large number of array data and therefore trigger their restriction.

After Cg shader is implemented, there are still some unpredictable issues. When no edit locator is added to toon shader, a large area of black block would appear on the shadow zone (seen as Fig.11). But after edit locator is applied, this area would return back to normal (seen as Fig.12). This situation does not only happen after applying editable shading function in shader, but also took place when completing the basic toon shading by using HLSL. Work is required

to figure out the issue behind it.

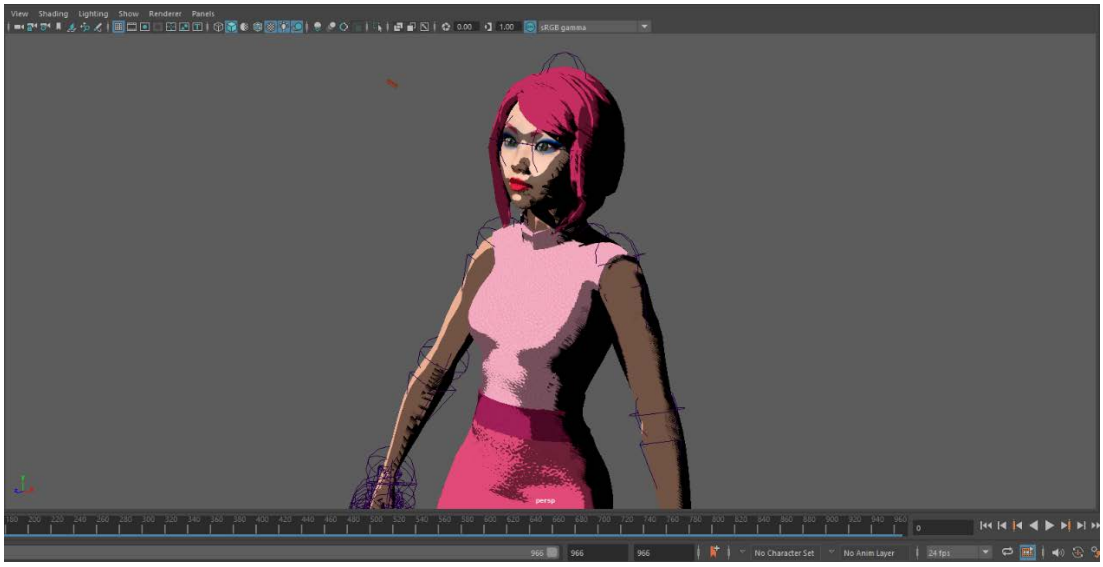


Fig. 11 Undesired dark block in shadow area (Cg & HLSL implementation)

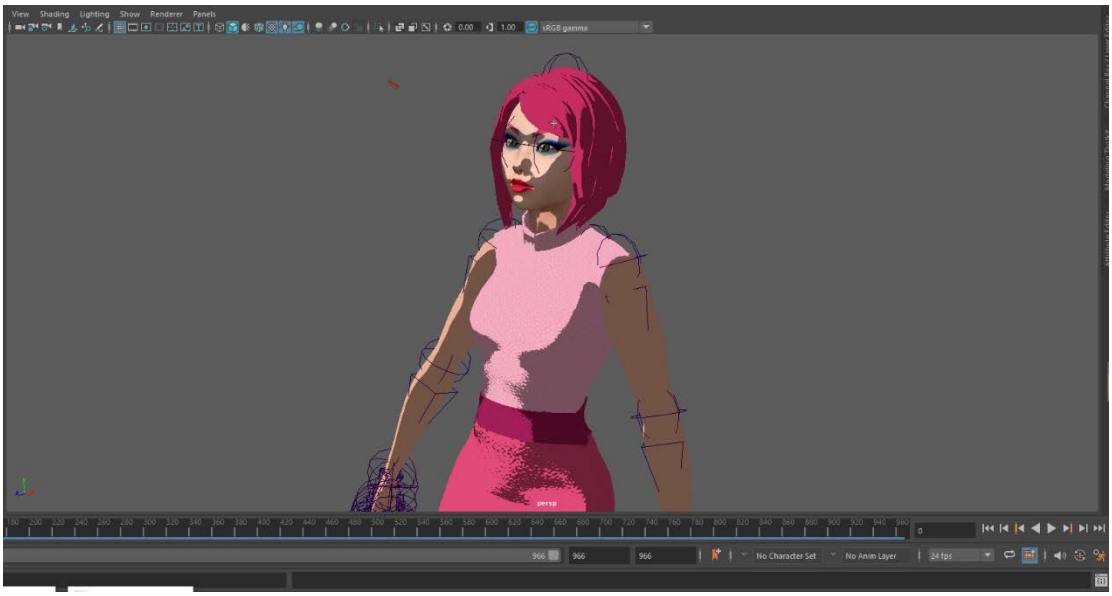


Fig. 12 Dark block disappear after adding edit locator (Cg implementation)

5. Conclusion & Future Work

Overall, the current outcome of the project does not achieve the initial objectives. The entire interactive tools built for workflow has been mostly finished and user can operate the shading tool in a convenient way. However,

editable shading effects can only appear in OpenGL Legency environment via Cg shader language and there are still many issues about algorithm implementation.

The majority development issue is that a technical solution cannot be found efficiently. Unclear documentations resulted in plenty of time was taken to review the code from development examples and do the tests. In order to make the shader running properly in OpenGL and DirectX environment, two possible solutions need to be highlighted:

1. Instead of writing raw text shaders in XML fragments, using scripts fragment commands (Autodesk Maya 2020) and shader files could work for large array data. The assumption is writing own shader could not define all the uniform data as parameters of main function of pixel shader, but the raw shader text compiled by OGS in XML fragment would do that.

2.Use multiple rendering passes instead of pass all the edit data into one shader. This approach is more feasible because it can highly reduce the pressure for data storage in shader. The only complex thing is that a rebuilding for the whole shader node is required.

These solution assumptions are proposed based on the debug test mentioned above. The issues were confirmed to be caused by too large array data passed to shaders, but still not confirmed if they were caused by the restriction of internal shader compiler built in XML fragment.

If this project was to be continue, the first thing would be to test algorithm and fix up the shading deviation. The debug process for algorithm implemented in shader could be complicated because no direct output is supported for shader. Running the shader code step by step or using a texture target output could be a better solution. In addition, keyframe interpolation for each edit could be finished as the main task is to ensure algorithm being implemented in a proper

way.

The development in Maya would still encounter lots of problems, which are hard to resolve if some technical details cannot be obtained from documentations directly. Therefore, technical support could help a lot to make the development more efficient.

References

Decaudin, P., 1996. Cartoon-Looking Rendering of 3D-Scenes. [online]. Available from: http://www-evasion.inrialpes.fr/people/Philippe.Decaudin [Accessed 21 Aug 2022].

Petikam, L., Anjyo, K. and Rhee, T., 2021. Shading Rig: Dynamic Art-directable Stylised Shading for 3D Characters. *ACM Transactions on Graphics*, 40 (5).

Sloan, P.-P. J., Martin, W., Gooch, A. and Gooch, B., n.d. *The Lit Sphere: A Model for Capturing NPR Shading from Art* [online]. Available from: http://www.cs.utah.edu/npr/papers/LitSphereHTML.

Todo, H., Anjyo, K., Baxter, W. and Igarashi, T., 2007. Locally controllable stylized shading. *ACM Transactions on Graphics*, 26 (99), 17.

Todo, H., Anjyo, K. and Yokoyama, S., 2013. Lit-Sphere extension for artistic rendering. *In*: *Visual Computer*. Springer Verlag, 473–480.

Tsai, Y. T. and Shih, Z. C., 2006. All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation. *ACM Transactions on Graphics*, 25 (3), 967–976.

Turk, G. and O'Brien, J. F., 1999. Shape transformation using variational implicit functions. *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1999*, 335–342.

Appendices

The Installation Instruction of Editable Shading Plugin

Use the python file in folder to install and uninstall the Maya plug-in

In Python3 environment, locate to the path of Maya2UnrealExportTool and run the following code in terminal:

```
python installModule.py
```

or

```
python installModule.py -d "{Your_Maya_Path}/maya"
```

The later one is for Maya installed in special machine whose location is not a default path.

To uninstall the plug-in, use the similar way:

```
python uninstallModule.py
```

After you install the plug-in, you can search it in Maya's Plug-in Manager (Windows > Settings/Preferences > Plug-in Manager).