Florian Ferstl
Marc Treib

Technische Universität München
Institut für Informatik
Lehrstuhl für Computergrafik & Visualisierung

Interactive Visual Data Analysis

Assignment 3 – *Iso-surface Ray Casting*

Finally, some actual visualization: This week, we will implement an iso-surface ray caster to visualize volumetric data sets. An iso-surface is composed of all points in the data set where the value is equal to a user-defined iso-value.

We'll also add a GUI.

3.1. GUI

Our tool should offer a simple graphical user interface to allow the user to change visualization parameters interactively.

Integrate the *AntTweakBar* library that you can find in `external/AntTweakBar_116` into your project. Remember not to copy any library files (lib, h, cpp, dll …) to your repository group folder but instead to access them directly using relative paths.

- Configure compiler and linker in the Visual Studio project settings to be able to use the library. Add a post-build step to automatically copy `AntTweakBar64.dll` to the output directory when the project is built.
- Integrate AntTweakBar by inserting the necessary `Tw*()` calls into the appropriate DXUT callback functions.
- To demonstrate everything works, create a `TwBar` and add some variables (as required in the following exercises).

Hint: Refer to the AntTweakBar documentation and tutorial on the official website and/or look at the code samples in `external\AntTweakBar_116\examples`.

3.2. Data

First, we have to get some volume data into your program. A few test data sets are available in `data/scalar`. Each data set consists of a `.dat` file and a `.raw` file. The `.dat` file contains metadata (such as data type, size etc.) in a `key: value` format and can be opened with any text editor, while the `.raw` file contains the actual binary data.

- Extend your `.dat` file parser from last week to also recognize `ObjectFileName`, `Resolution` and `Format` (which, for now, should always be `BYTE` or `UCHAR`); you can ignore any other keys. Also, note that `SliceThickness` actually makes sense now: It corresponds to the grid spacing per dimension. The size of the data set in object space is `Resolution*SliceThickness`!
- Load the binary data from the corresponding `.raw` file specified by `ObjectFileName`, according to the `Resolution` and `Format`.
- Create a 3D texture resource using `ID3D11Device::CreateTexture3D` and fill it with the data. Also create a corresponding shader resource view.

Hint: Refer to the D3D11 documentation in the MSDN for information on `CreateTexture3D` and `CreateShaderResourceView`. Remember to check the return values (in the code) as well as Visual Studio's Output window in debug mode (manually) to catch any errors!

3.3. Ray Casting Box

Render a solid box (6 faces × 2 triangles) with the same size as the bounding box.

- Add 3D texture coordinates $\in \{0,1\}^3$ to the vertices of the box.
- Make sure all triangles are oriented consistently, i.e. the result should look the same whether you enable back face culling or not.
- For a start, use the texture coordinates to color the box (mapping xyz to rgb) so that the corners of the box are white, black, red, green, blue, yellow, pink, and cyan. We recommend two debug shaders:
  - *Debug Shader 1*: Use **back** face culling and xyz-to-rgb coloring to render the **front** faces of the box.
  - *Debug Shader 2*: Use **front** face culling and xyz-to-rgb coloring to render the **back** faces of the box (which will look a bit weird).

For ray casting, we will need both entry and exit point in the same pixel shader. We will compute the entry point analytically, using the "color values" from *Debug Shader 2* as the current ray's exit point from the box.

- Write a *Basic Ray Casting Box Shader* that computes the first intersection of the ray (camera position to exit point) with the box. Hint: You will need the camera position in object space!
- For now, transform this entry position into the range $[0;1]^3$ as well and use it as the pixel color.
- The result of your *Basic Ray Casting Box Shader* should now look identical to the result of *Debug Shader 1*. **Make sure this works correctly** – it'll cause you no end of trouble later if there is a bug here! It might make sense to temporarily add GUI controls to quickly switch between the shaders.

3.4. Ray Marching

Extend the ray casting pixel shader to perform the actual ray marching in the 3D texture that you created: Go from the bounding box's entry point towards the exit point with steps of a fixed size. At each step, sample the 3D texture and compare the result with the iso-value. Stop when you either find a value larger than or equal to the iso-value, or you reach the exit point.

- If the size of the volume is not the same in all dimensions, adjust the box accordingly.
- Add controls to your UI to specify the step size and the iso-value. A reasonable default value for the step size is half the distance between two grid points (i.e. `0.5/texture_size` in texture coordinates).
- For now, just return a fixed color if you have found a point on the iso-surface. If you reach the exit point without hitting the iso-surface, `discard` the fragment.

You should now see the silhouette of the surface, and be able to change it by adjusting the iso-value.

3.5. Illumination

The solid-color iso-surfaces do not illustrate the shape of the object and, more importantly, they look boring! We will now add Phong lighting to the surface to remedy this.

- To compute lighting, we need a normal vector. The normal vector of an iso-surface can be approximated by the (normalized) negative gradient of the volume data. When you have found an iso-surface in your shader, compute a gradient using central differences and normalize it to get the surface normal vector.
- Use this normal vector to compute Phong (or Blinn-Phong) lighting, including ambient, diffuse and specular terms. Add controls to your UI to control the material color and shininess (i.e. specular exponent). As in the previous assignment, you can use a head light, i.e. light position = camera position.

3.6. Intersection Refinement: Binary Search

Unless you choose a very small step size, your illuminated iso-surface will have severe banding artifacts. This is because we have not computed the actual intersection between the ray and the iso-surface so far – we might be off by as much as one step! To fix this, we'll use a binary search to narrow down the actual intersection point.

- When we have found the first value larger than the iso-value (as in Assignment 3.4), we know that the iso-surface lies between this point and the previous step.
- We'll use a binary search to get closer to the actual intersection: Start with the interval `[a,b] = [last_pos,cur_pos]` and check the value at the middle position `(a+b)/2`. If it's larger than the iso-value, the intersection has to be in the first half of the interval, so continue with `[a,(a+b)/2]`. Otherwise, continue with `[(a+b)/2,b]`.
- For simplicity and performance, just use a fixed number of binary search steps (4 or 5 is usually okay).

The working solution must be committed till **November 6, 09:00am**. If anything is not working as described here or if you want a specific SVN revision to be rated, explain yourself in the `readme.txt` file within your `solution` directory.