SUNG-EUI YOON, KAIST

RENDERING

First printing, July 2018

5 Interaction

In this chapter, we discuss basic ways of interacting with 3D objects. We first discuss a file format of 3D objects (Sec. 5.1), and how to select and manipulate those objects (Sec. 5.2). We then discuss a simple way of supporting 3D rotation based on a concept of the virtual trackball (Sec. 5.3), followed by handling hierarchically defined models (Sec. 5.4).

5.1 Loading Objects

One can create a 3D object using various modeling tools such as Blender, a free and open-source software, and Autodesk 3ds Max, a commercial tool. Also, many 3D models have been created and available commercially and freely at various websites. As a result, it is also common to load those models and compose a 3D scene with them.

As a step to compose and render such a scene, it is necessary to read and load 3D objects. Many file formats are proposed to enable such operations easily. In this section, we discuss an obj format, one of simplest and widely available formats. A simple example of an obj file format is shown in Frame 5.1.

```
# A simple cube in an obj file format          // strings starting
with # are comments
    v 1 1 1                                    // vertex specification
    v 1 1 -1
    v 1 -1 1
    v 1 -1 -1
    v -1 1 1
    v -1 1 -1
    v -1 -1 1
    v -1 -1 -1
```

```
f 1 3 4                              // face specification
f 5 6 8
f 1 2 6
f 3 7 8
f 1 5 7
f 2 4 8
```

Basic obj file tokens are explained in below:

- **# comments.** The rest of the line starting with # is comment.
- **v float float float.** It specifies X, Y, and Z coordinates of a vertex.
- **vn float float float.** It defines a normal.
- **vt float float.** It specifies U and V texture coordinates.
- **f int int int ..** It defines a triangle (or other polygon) with vertices with specified indices. These arguments are 1-based indices. When we do not have normal information associated with the triangle, we compute the normal out of the plane passing the triangle. The direction, i.e., inward or outward, of the normal vector depends on the ordering of those vertices (Ch. 6.3). As a result, an extra attention is required on the ordering of vertices.

We can also read and store these files in an ASCII mode or binary mode. It is usually more intuitive for human to use the ASCII mode, since we can effective understand what the file describes. On the other hand, the binary mode has benefits in terms of compact storage and thus fast I/O operations.

Layouts. One can have an arbitrary ordering, i.e., layout, of vertices and triangles. Nonetheless, the layout has been identified to play an important role in terms of performance. Since modern computer architectures adopt a block-based cache, the cache fetches a block containing consecutively located data, when one of those data is accessed. As a result, data that are likely to be accessed together are recommended to be stored closely. This idea leads to cache-coherent and cache-oblivious layouts [1]. .

[1] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-Oblivious Mesh Layouts. *ACM Transactions on Graphics (SIGGRAPH)*, 24 (3):886–893, 2005

5.2 Selection

To interact with objects in graphics applications, we first need a way of selecting a particular object in the 3D scene. Suppose that we would like to select an object that the mouse pointer is locating at.

Many possible approaches are possible, and two of them are listed here:

1. **Object-space approach.** Given the point of the mouse cursor, we can imagine a virtual ray passing from the camera origin to the point. We can then identify objects that are intersecting objects and choose the object that has the closest intersection point to the viewer. Overall, this approach is ray casting, which is the basis for ray tracing, a critical component of physically-based rendering (Ch. 10).

2. **Image-space approach.** Since we have a rendered image in the color buffer, we can directly access the pixel in the buffer, where the mouse cursor is located at. Unfortunately, the pixel has only the color of a rendered triangle, not the ID of the triangle. We explain a concept of an item buffer that encodes the ID of each triangle based on a color. This approach unlike the object space method based on ray tracing works on the image buffer and thus is an image-space approach.

It is worthwhile to mention that many graphics problems can be approached in either the object-space, image-space, or even a hybrid approach combining both of them, as described for the selection problem.

5.2.1 *Selection with Item Buffer*

For the selection problem mentioned in the prior section, we want to encode a triangle ID on each pixel on a buffer.

A simple way given the rendering pipeline is to use the concept of the item buffer. The item buffer is simply a different name to the color buffer, with the difference of encoding IDs of triangles, not the original colors of them. To encode an ID for each triangle, we use a unique RGB color value, ID color, for each triangle or each object that serves a smallest selection granularity.

We render all the objects with those ID colors, but we should not show this result to a viewer, since this is not the final result. Therefore, we render it to a back buffer, but do not swap it to the front buffer that is accessed by a display device and thus visible to the viewer. We then read the back buffer by calling an appropriate access function, e.g., $glReadPixels(\cdot)$. Once we fetch the color ID given the chosen pixel, we can identify its associate triangle or object. We then provide a feedback based on the selection operation and render the scene with its original colors.

Note that this selection method works by reading the buffer and thus is categorized by an image space approach. As a result, this

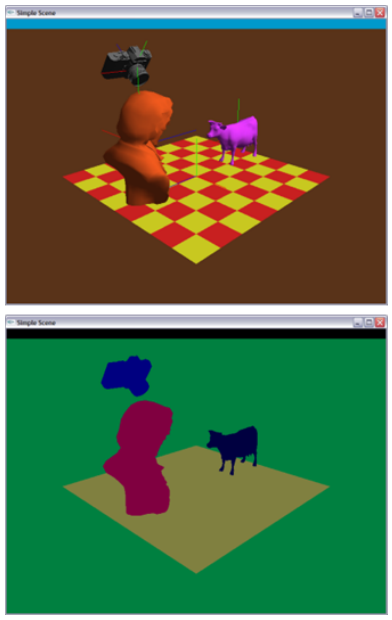
Figure 5.1: The top image shows the color buffer of a scene, while the bottom image shows its item buffer rendered in the back buffer.

method shows common features of many image space approaches, and some of them are them are:

1. Accuracy depending on the image resolution. A main characteristic of the image-space method is that its accuracy depends on the chosen image resolution, since we identify the triangle ID by accessing an pixel in the item buffer. When we have multiple triangles in a pixel, the pixel can encode only a single triangle. As a result, as we have higher resolutions, we have a higher accuracy in terms of identifying a chosen triangle. Note that when we choose a triangle based on a ray in the object-space method, we do not have such a characteristic.

Accuracy of image-space methods are commonly controlled by the chosen resolution of images.

2. Different performance characteristics to the object space approach. While the image-space method has its accuracy issue, it is commonly used in many different problems including the selection problem, since it is relatively easy to implement and to show high performance, mainly thanks to the support from GPUs. For example, we render triangles and read the buffer through GPU, and thus they can be done quite quickly. Nonetheless, it is less obvious whether this approach has a better time complexity. Specifically, the image-space method using the item buffer explained in this section has a linear time complexity, while the ray tracing based approach using an acceleration structure such as bounding volume hierarchy has sub-linear complexity (Ch. 10.3). As a result, when we have many objects and triangles, the object-space approach can be faster.

In this section, we studied about an image-space selection method using the item buffer. More importantly, we discussed its different characteristics with those of an object-space method using ray tracing.

5.3 *Virtual Trackball*

In the prior section, we discussed how to pick an object. Once we select an object, we would like to re-position or re-orient the object. For such operations, we can do that through many input devices such as keyboard, mouse, touch screen, etc. For example, many modeling tools (e.g., Autodesk 3ds Max) provide various object and camera manipulations through mouse, which is an inexpensive and widely used input device.

Figure 5.2: A trackball. The image is excerpted from the homepage of its vendor, Kensington.

Discussing various interaction operations with available input devices is beyond the scope of this section. Instead, we focus on how to rotate an object in a 3D space. Fig. 5.2 shows a trackball, where a rolling ball is attached. We can use the trackball to intuitively rotate an object, which is mapped to the ball on the track ball. Unfortunately, the trackball is not widely available compared to keyboard and mouse. We now see how we can support such convenient rotation mechanisms with a mouse.

Suppose that we enclose a sphere on an object that we would like to rotate. Fig. 5.3-top image shows such a configuration. The 2D grid represents our viewing plane. The interaction scenario for rotating the object with the mouse works as the following: 1) the user locates the mouse cursor and clicks a button at a point, $\vec{a}$ [2], and then move and specify the cursor into a different position, $\vec{b}$. Basically, based on this interaction scheme, we want to roll the ball from $\vec{a}$ to $\vec{b}$. The next question is how to compute rotation information, the rotation axis, $\vec{r}$, and its rotation amount, θ, realizing the rolling operation?

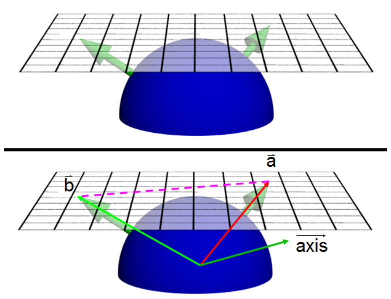

Figure 5.3: Top: we place a ball on an object under the rotation, while the viewing space is touching the ball. Bottom: suppose that we push a button of a mouse at the location of $\vec{a}$ and release it at the another location, $\vec{b}$. In this user input, we want to rotate the ball and its enclosed object from $\vec{a}$ to $\vec{b}$.

[2] We represent this as a vector starting from the ball origin to the point.

Suppose that you grasp the ball from $\vec{a}$ to $\vec{b}$ in your right (or left) hand. The thumb in this case indicates the rotation axis $\vec{r}$. The rotation axis is a vector orthogonal to both $\vec{a}$ and $\vec{b}$, and this can be computed by the cross product between them:

$$\hat{r} = \hat{a} \times \hat{b}, \tag{5.1}$$

where $\hat{r}$ represents a normalized vector whose magnitude is one, i.e., $\hat{r} = \frac{\vec{r}}{|\vec{r}|}$. The rotation angle θ is computed by the inverse of the dot product:

$$\theta = \cos^{-1}(\hat{a} \cdot \hat{b}). \tag{5.2}$$

If necessary, we can also compute a rotation matrix based on computed axis and angle (Sec. 3.5).

5.4 *Transformation Hierarchy*

Some objects have many joints (Fig. 5.4), and we can move each joint independently. In this section, we would like to compute transformations for those parts of an object.

As an example for the study, let's consider an object consisting of two parts with a joint (the rightmost object in Fig. 5.4). Each part is usually defined in its own modeling coordinate; its center is commonly located at the origin, say $(0, 0, 0)$, in its modeling coordinate. We then apply appropriate transformations to those parts to locate it at the world space. Since these parts are defined hierarchically, these transformations are also defined in the same, hierarchical way.

Suppose that $\mathbf{M_b}$ and $\mathbf{M_p}$ are two transformation matrices that converts from the base to the world, and from the part to the base, respectively. In this context, to compute the base, say, its coordinate b in its modeling space, in the world, we compute such transformed locations based on $\mathbf{M_b} b$. For the part, p, we need to apply $\mathbf{M_p}$ to

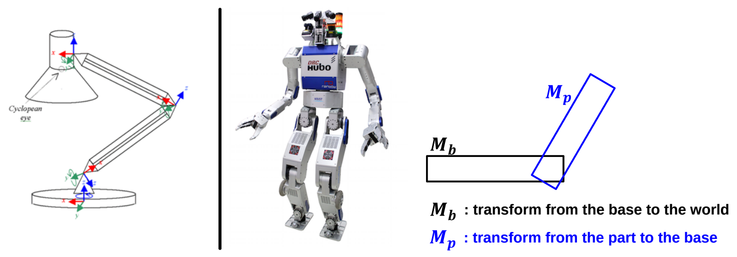
Cyclopean eye
DRC HUBO
M_p
M_b
M_b : transform from the base to the world
M_p : transform from the part to the base

Figure 5.4: The left and middle show two examples of objects with many joints. The left is a lamp with many joints, and the middle shows a humanoid robot, DRC hubo from KAIST, who won DRC (DARPA Robotics Challenge) held at 2015. The right shows an example object consisting of two parts.

locate the part in the base space, followed by $\mathbf{M_b}$ to the world space. As a result, we apply $\mathbf{M_b}\mathbf{M_p}p$.