3 Method

As was discussed in Chapter 2, the majority of artists uses professional (often commercial) 3D software tools. We want to improve their workflow by using custom real-time renderer.

3.1 Motivation

We see two issues standing in the way between illustrators and animators, and more effective workflow. We believe that by using the real-time renderer cellVIEW, we can solve both of them.

First problem stems from the fact that modern professional 3D programs are designed primarily for different scenarios than scientific, specifically molecular, content. They mostly use polygon meshes as the representation of 3D scenes. Mesh is a set of vertices, edges and triangles used to model real-life objects. However, for molecular data and scenes, meshes are not the best representation. A single atom is in our case simplified by a single sphere of certain radius. Approximation of a sphere using meshes is, unfortunately, not very good in terms of how many triangles are needed for a smooth sphere. Very detailed mesh (consisting of hundreds of triangles) must be used to create an effect of smooth spherical object (see Figure 3.1). Given the fact that our scenes may consist of milions of atom, this creates a big problem in terms of memory space requirements.

Other representations and rendering techniques have been created to render molecules and atoms. Billboard, sometimes also called im-

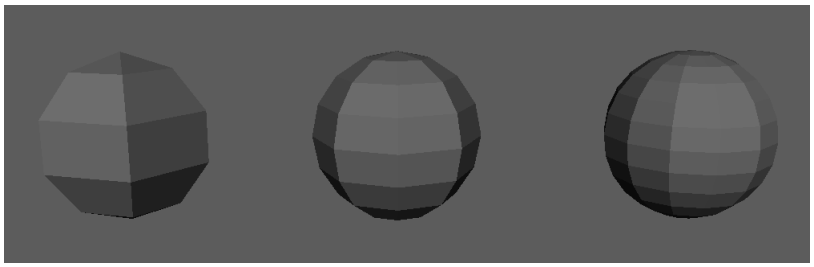

Figure 3.1: Spheres with different level of detail. From left: 40 triangles, 112 triangles, 264 triangles

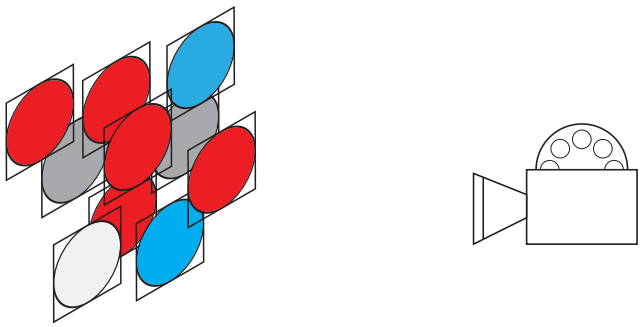

Figure 3.2: Molecule rendering via impostors

postor, rendering technique is considered to be the most efficient way of rendering atomic data today (see Figure 3.2). Instead of full sphere geometry, we only render a quad for each atom. This quad is then made to always face the camera. The illusion of sphere is acomplished by using a fragment shader that draws the sphere as if it was projected into this quad.

Second issue that we wanted to address is an issue of efficiency. Off-line rendering takes some time to finish. When artist is working, the rendering step is holding them back. It kicks them out of the *state of flow* [14]. To address this issue, we want to provide the artist with an immediate feedback. Every change he or she makes in the scene should be immediately reflected in the resulting output.

The benefit of using a real-time renderer plays into our cards [TODO: preformulovat]. We already have a reason to use cellVIEW as a renderer (because it is better for the data we deal with). Because it is real-time (immediate), we just need a way how to utilize this property.

In the end, we would want to completely eliminate the step of rendering and make the process of visual feedback as seemless as possible (as illustrated in Figure 3.3).

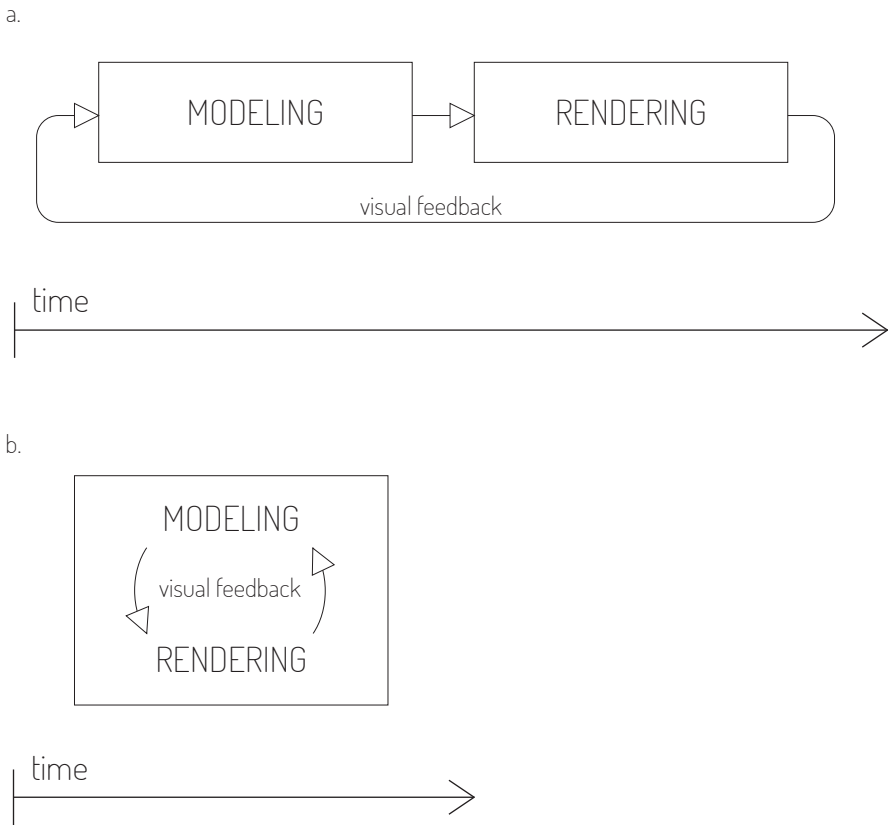

Figure 3.3: Workflow, a. traditional approach with off-line rendering, b. our approach with real-time rendering. By using more simple, faster rendering method, we eliminate the rendering step completely

3.2 Immediate Feedback via Shared Memory

Ultimately, we have two programs and we want to use some functionality from the first one and other functionality from the second one. The naive solution would be to implement our desired features into the software tool that is missing it. In our case, that would mean we could either implement a fast and visually appealing renderer into the modeling software tool, or we could do the opposite and implement the desired modeling tools into the renderer program.

The problem with the first approach is that substantial percentage of the 3D software packages that artists use the most are commercial solutions with closed source code. It is possible to extend them via API that they provide but that is not enough if we want to implement state-of-the-art technique that requires the latest technology in terms of graphics API etc.

We also do not want to implement desired 3D modeling and animation tools into the specific molecular visualization program. This would create development overhead which would not bring much benefit on its own. Besides that, every artist uses different instruments (different software, shortcuts, additional plug-ins, etc.) and pleasing all of them would be an impossible goal to achieve.

Instead, we have chosen to go with another, third, option. We do not want to re-implement from scratch something that is already available to use. Instead, we went for a different approach and tried to connect the two parts in a way that would allow us to use both sets of features at the same time. We do this by using both of the programs and establishing a communication channel between them.

We use writing and reading from shared memory to accomlish this communication. In our case the communication is one-way. We can model or animate the scene in one program and transfer the data describing this scene to have it rendered in the second program. As we will show, this can be done in a straightforward fashion as our scenes can be simplified to the point of describing them with a few numbers per atom.

On the side of modeling program, we want to see an approximation of the scene. One way would be to use a simple geometry like cube or sphere as a placeholder for each molecule. Such placeholder would then serve as an object that we manipulate instead of an actual

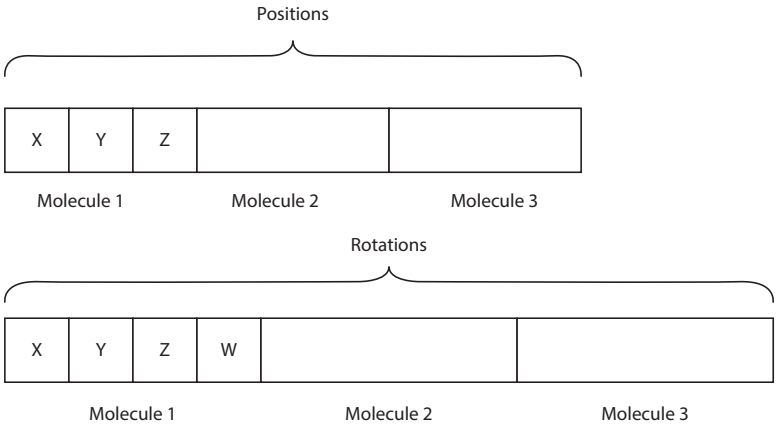

Figure 3.4: Shared memory data layout

molecule. We tried something a little bit different. With Molecular Maya plug-in, we can load a molecule description in the form of a PDB file. With this plug-in it is also possible to generate a polygonal surface mesh representation with choosable detail level. We therefore load a molecule and generate a very low resolution mesh which serves as our placeholder.

3.3 Transferred Data

Thanks to the standardization in the form of PDB files which are stored in a central database, we are able to decrease the amount of data we need to transfer between programs. A typical molecular scene consists of macromolecules: proteins and lipids.

Two simplifications are used. First, we consider the shape of the molecule to be static. This means that inside each molecule, the positions of its atoms do not change in time. These positions are taken from the PDB file. Second, all the instances of a certain molecule look the same. Individual instances of a certain molecule type only differ in their positions and rotations.

With these facts in mind, the molecular scene can be defined by three buffers (arrays):

- Positions buffer: XYZ positions of all the instances in the scene
- Rotations buffer: rotations of all the instances in the scene, represented quaternions
- Info buffer: containing information about type of each instance

Figure 3.4 shows the layout of the data. Because of technical implications, all positions are grouped together, same with rotations and informations, instead of grouping all the information (position, rotation, type) about instance.

Besides the actual scene data, we also want to use shared memory to synchronize cameras on both endpoints. The feel of connection would not be complete if the artist had to control two cameras (one in Maya and one in cellVIEW) at the same time. Because of that, we want to reflect the camera movements from Maya into cellVIEW.

To do that, we need to share position and rotation of the camera in Maya. Another shared memory segment is allocated just for this data. On cellVIEW side, we then read the position and rotation of camera in Maya and set the main camera in cellVIEW to have the same parameters. This way what the user looks at in Maya, he sees from the same point of view in cellVIEW.

3.4 Limitations

The obvious limitation is the amount of space that can be allocated in shared memory. As we will see in Chapter 4, we can use shared memory that is backed by paging system. This means that, theoritically, we should be able to allocate very large chunks of memory. However, in case of working with a very large data sets, the speed of writing into and reading from shared memory might not be ideal.

Another limit could be the amount of placeholder geometry that Maya can render in its viewport.