UNIVERSITÀ DEGLI STUDI DI TRIESTE
Dipartimento di Ingegneria e Architettura

Laurea Magistrale in Ingegneria Informatica

Design and implementation of a Virtual Reality application for Computational Fluid Dynamics

Dicembre 2017

Laureando
Lorenzo D'Eri

Relatore
Prof. Francesco Fabris

Anno Accademico 2016/2017

large availability of pre-built community-made tutorials and tools, and the inherent cross-platform compatibility offered by Unity, the decision of proceeding in this direction seemed the most appropriate.

3.2 Application Architecture

The application's architecture has been designed to be as flexible and expandable as possible, so that any additional feature could be added to the VR environment with minimum effort.

The project consists in only one scene, called `Main scene`, in which all the objects are inserted with the hierarchy shown in Figure 3.2. Each feature is controlled by **Managers**, which implement the business logic of the application through behavior scripts (as further detailed in Section 3.3). Managers communicate with each other through references, and are designed to expose a simple and coherent interface to the upper layers of the application.

As stated in Section 2.2.3, Unity offers two different programming languages for the behavior scripts: C# or UnityJava. For the purpose of this project, all scripts have been developed in C#.

Some managers have children objects, which are usually the actual models rendered in the Virtual Reality environment. It is important to notice that the parent-child relationship between the managers and their children has nothing to do with the concept of inheritance in programming. In fact, defining a parent in Unity simply means that the child's position, rotation and scale will be relative to the parent's; this is especially useful to handle animations (as seen for example in Section 3.3.5).

Inter-object communication happens through a set of globally shared variables. In fact, although Unity provides ways of transversing the scene in order to look for specific objects, this has a non-negligible computational cost, and is often advised against by the community. Instead, each object that needs to be accessed from outside registers itself to a `Global` static class. Then, at runtime, the application

begins with an initialization process in which each manager stores the references to the objects it needs to communicate with as private fields, for faster subsequent accesses. On the other hand, all non-manager objects need not know about the external parts of the software, in accordance to the SOLID principles of Object-Oriented Programming (as described in [25]).

Internally, both Managers and objects have one or more behavioral scripts attached, in which the overridden implementation of `Awake`, `Start` and optionally `Update` are defined. Furthermore, methods have been implemented to trigger and process custom events, such as the loading of an object from ParaView. Such methods follow the naming convention `OnEventName`, and are called through a callback system.

3.2.1 Top-level Managers

At the current stage of development, the following top-level managers have been defined, as seen in Figure 3.2):

- **VRManager**: core of the *VRTK* plugin (see Section 2.2.3.2), responsible for the Virtual Reality hardware (sensors, controllers, camera) and their 3D representations.
- **ModeManager**: exposes utility methods to check whether the Unity application is running as a stand-alone executable ("Player mode") or inside the Unity editor ("Editor mode"). Used by **ParaViewManager**.
- **EnvironmentManager**: responsible for the virtual environment in which the user is immersed, i.e. the room. It exposes methods to toggle the room's visibility, used by **StaticMenuManager**.
- **StaticMenuManager**: manages the static menu that the user can summon by pressing the Menu button on the controller. It's responsible for toggling its visibility and for binding the menu buttons to the appropriate functions.
- **LightManager**: responsible for the lighting of the environment. Currently exposes a function to set the intensity of the ambient lighting.

Currently, the user can adjust the brightness of the environment via a slider on the static menu. The function that implements this is:

```
public void SetIntensity(float value) {
  RenderSettings.ambientLight = new Color(value, value, value, 1);
}
```

3.3.3 ParaView object loading

As stated before, the most important feature is that the application must be able to communicate with ParaView, load its objects and render them in the virtual world. In the current implementation, this task is handled by a `ParaviewObjectLoader`. This is responsible for the following:

- Setting up a *TCP listener* for incoming connections from ParaView,
- Implementing the communication protocol for import/export synchronization (see Section 4.3.2),
- Reading the data from memory,
- Calling the appropriate methods to construct the 3D model from the data,
- Registering the object as global and triggering appropriate events.

As this is the only scenario that is highly dependent on external input (i.e. an object being send from ParaView, an external application), the design of the object loader required careful handling of its asynchronous nature. For this purpose, a custom event named `ParaviewObjectLoaded` has been designed, in order for the other manager to register their callbacks. This is important as some code (for example the autoresizing of the object) must only be executed after an object has been loaded from ParaView, which could be in any moment after the beginning of the application, and does not depend on the application itself.

A similar consideration can be made for the object's destruction. In fact, supposing that the user wants to substitute the currently loaded object with another one, the first must first be deleted. In fact, currently the application supports only one object being loaded at any time. This is achieved through another event, called `ParaviewObjectUnloaded`, which is necessary for the other managers to clean up any reference to the object and to update their state before importing another one.

As an example, the code to register a loaded ParaView object and call the appropriate callbacks is:

```
public static void RegisterParaviewObject(GameObject paraviewObj) {
  // Register object in globals
  Globals.paraviewObj = paraviewObj;

  // Trigger callbacks
  if (ParaviewObjectLoadedCallbacks != null)
    ParaviewObjectLoadedCallbacks(paraviewObj);
}
```

3.3.3.1 3D model creation

To import an object this must first be read from memory; in its current implementation, data is stored in RAM in X3D format (see Section 4.3.2). This means that in order to read the data the application must make the appropriate calls to the *Windows API*, and subsequently pass the result of this operation to an entity that can convert X3D strings into 3D models.

The reading phase is carried out by an object that exists at a lower level of abstraction: the `XDocumentLoader`. This exposes a `Load()` function, which takes the name and the size of the object as arguments and returns a properly formatted `XDocument`, (a XML representation of the object). The `Load()` function is implemented as follows:

```
public static XDocument Load(string objectName, uint objectSize) {
  // Attach to the object in shared memory
  sHandle = new SafeFileHandle(hHandle, true);
  Attach(objectName, objectSize);

  // Parse the raw data into XML
  XDocument doc = XDocument.Parse(pBuffer);

  Detach();
  return doc;
}
```

The `XDocument` is then passed to an external library, called `X3DLoader`, which handles the conversion.

3.3.3.2 3D model material

One of the most important aspects of CFD data visualization is coloring. In fact, more often than not, it can be extremely useful to color an object according to the value of a certain variable (such as pressure, temperature, ...) on each face of the mesh.

Such information about the color is included in the X3D representation of the object, but must be rendered in Unity through the use of *shaders*. For this purpose, a custom shader that implements **Vertex Color Shading** has been included in the project.

In order to programmatically apply a material based on such shader to the object, the design choice has been to include a non-visible object, called `MaterialPlaceholder`, from which the `ParaviewObjectManager` can read information about the material chosen by the developer and apply it on any loaded objects. This gives freedom to customize the material through the editor, even when no Paraview objects are loaded in the scene.

Figure 3.5 shows an example of a mesh colored according to pressure, blue being lower values and red being higher values.

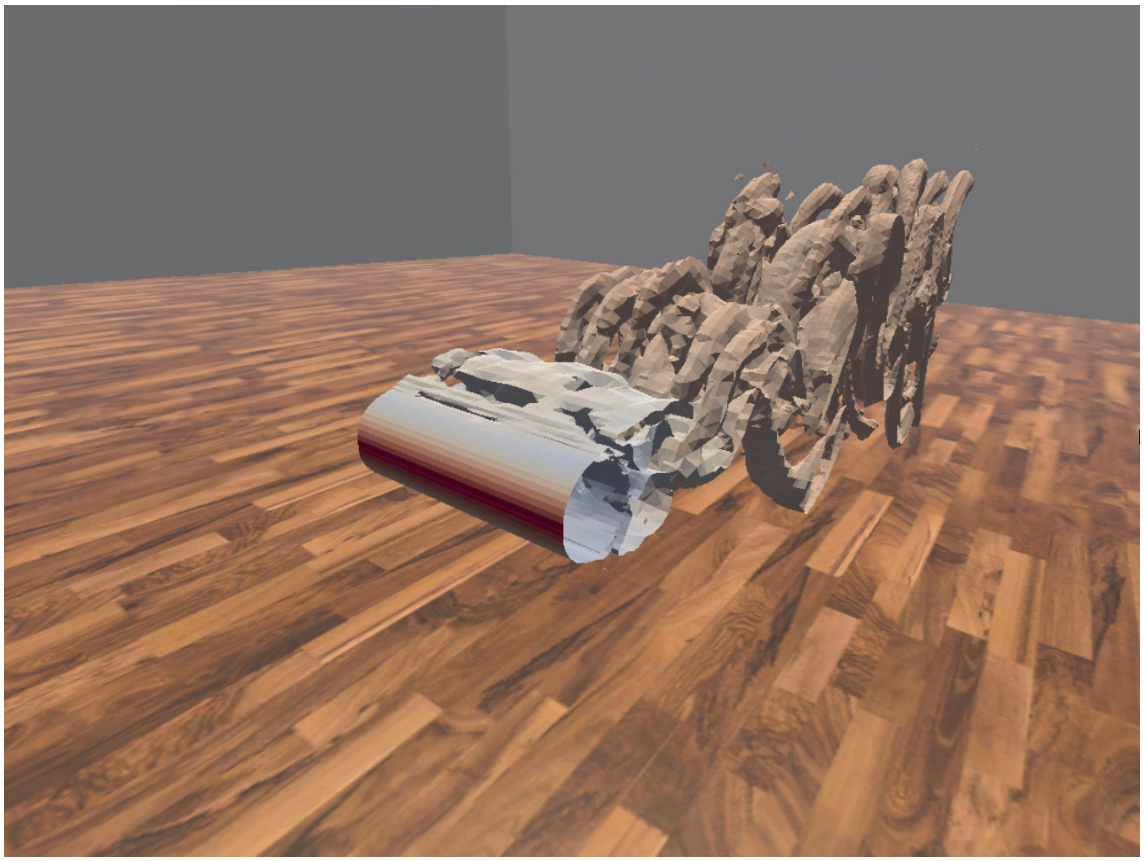

FIGURE 3.5: Screenshot of a mesh colored according to its values

3.3.4 Static Menu

As seen in the previous sections, some of the aspects of the VR environment can be changed at runtime. This happens through a menu, which is called `StaticMenu` to distinguish it from the radial menu (see Section 3.3.5).

By default the menu is hidden, so as not to interfere with the object visualization, but can be toggled through the *Menu button* on any of the two controllers. The behavior defined in the `StaticMenuManager` class mimics that of the SteamVR dashboard, which is the default system menu of the whole SteamVR environment: when opened, the static menu positions itself at a fixed distance directly in front of the user, facing them. If then the user moves around, the menu stays in its place in the 3D world until hidden and shown again. By usability testing, this has proven to be the most natural and unobstructive solution for the users.

FIGURE 3.7: Screenshot of the opened radial menu

3.3.6 Object interaction

Among the project objectives outlined in Section 1.1, one of the most important was the ability to interact with ParaView's object. In fact, this is essential for any true Virtual Reality application: in the real world objects can be touched, moved, rotated and manipulated. Therefore, when a user enters a VR environment, they naturally expect the same laws to apply, in which case they feel more immersed. Moreover, one must bear in mind that the scope of the application discussed in this thesis is the analysis of CFD data, and therefore the ability to manipulate the dataset in a 3D environment is an enormous advantage over the classical visualization on a 2D screen.

Bearing this in mind, basic form of interaction with the model have been implemented, with two exceptions: gravity and solidness. On one hand, having the model

react to gravity is impractical and counter-productive, as it means that it cannot be left floating at eye level for detailed examination; a similar consideration applies to solidness: given that with current technologies it is not possible to simulate touch through the controller, the most commonly accepted way of grabbing and object is by keeping a button pressed (the trigger, in this case) while holding the controller inside of it.

Apart from this, the object is designed to behave like a rigid body, meaning that it can be grabbed by any of its parts, and it can be translated and rotated in 3D space.

To achieve this, the object is given an `Interactable` behavior at loading time. This, internally, gives it rigid body properties, and defines a collider. In Unity, a collider is a region surrounding the object that represents its boundaries, for the purpose of determining collision. As an object enters or exits another object's collider, an event is triggered. This can be bound to appropriate callbacks to implement grabbing, as in the case being discussed. For performance reasons, the collider's shape for the object was chosen to be the minimum bounding box (see Section 3.3.6.1).

The code for implementing the object grabbing lies both in the controller and in the interactable object itself, although the object's end is more interesting. In fact, whenever a controller is inside the bounding box, and the trigger is pressed, the method `OnBeginInteraction()` is called. Similarly, when the trigger is released, the method `OnEndInteraction()` is invoked.

```
public void OnBeginInteraction(Controller controller) {
  // The object becames integral with the controller
  this.transform.parent = controller.transform;

  // Save reference to the controller
  attachedController = controller;
}

public void OnEndInteraction(Controller controller) {
  if (controller == attachedController)
  {
    // Release the object
    this.transform.parent = null;

    // Reset reference
    attachedController = null;
  }
}
```

It is worth noting that saving and checking the reference to the controller is necessary to implement the possibility of passing the object from one controller to the other without releasing it. In fact, one can see how if a second `OnBeginInteraction()` is called from the other controller before the interaction with the first ends, the reference is updated. Therefore, when the trigger of the first controller will be released, the object will still be integral with the second controller, as expected.

3.3.6.1 Bounding box

As explained before, each ParaView object is surrounded by a bounding box that acts as a collider. This allows interaction with the controllers.

It is worth mentioning, however, that the fact that the bounding box is effectively bigger than the mesh itself can be counter-intuitive for the user. In fact, it means that there are regions of space that are outside the mesh but inside the bounding box, thus making the object grabbable without the controller being inside of it. To limit the effects of this, and ease the process of grabbing, an option on the static menu lets the user enable a visible bounding box. This is rendered as a semi-transparent

box, as seen in Figure 3.8 and is shown only when the controller is actively colliding with it.

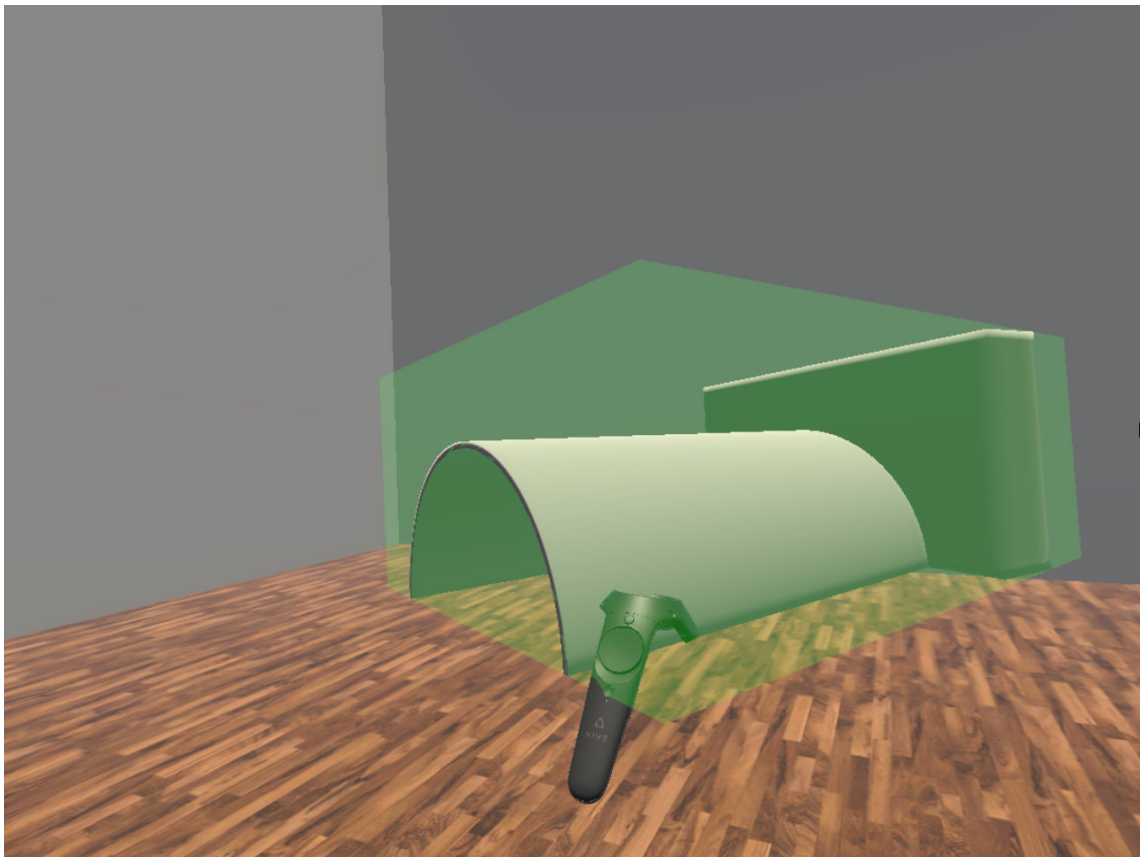

FIGURE 3.8: Screenshot of the bounding box

Another feature of the current implementation of the bounding box is its ability to adapt in the case of animated models. In fact, in each frame of the animation the model can have a different size and shape, thus requiring a different bounding box. To solve this problem, it has been chosen to implement a function that actively resizes the bounding box depending on the current shape of the object: such function is bound as a callback to the `NextFrameLoaded` event, which is triggered by the animation manager after every frame (see Section 3.3.8). The code that resizes the bounding box is the following:

```
public void FitColliderToChildren() {
  // Temporarily reset the parent's orientation, since the bounds will
  // be given in world coordinates but the collider is in local coordinates.
  Quaternion oldRotation = transform.rotation;
  transform.rotation = Quaternion.identity;

  bool hasBounds = false;
  Bounds bounds = new Bounds(Vector3.zero, Vector3.zero);

  // Iterate over all active children.
  foreach(Renderer childRenderer in GetComponentsInChildren<MeshRenderer>()) {
    if (hasBounds)
      bounds.Encapsulate(childRenderer.bounds);
    else {
      bounds = childRenderer.bounds;
      hasBounds = true;
    }
  }

  // Set collider center and size (with conversion from global to local)
  BoxCollider collider = GetComponent<BoxCollider>();
  collider.center = Scale(bounds.center - transform.position, transform.
  ↪ localScale.Reciprocal());
  collider.size = Scale(bounds.size, transform.localScale.Reciprocal());

  // Reset the objects orientation
  transform.rotation = oldRotation;
}
```

This function is a good example of the concept of global and local coordinates in Unity. In fact, the bounding box is considered as an independent object, and therefore expressed in global coordinates, while the Paraview Object itself is a child of `ParaViewManager`, and therefore it uses local coordinates relative to its parent. For this reason, some of the entities involved require transformations before being applied.

3.3.7 Resizing

In general, the size of CFD models can span over several orders of magnitude depending on the object being studied: from micrometers in the case of the flow of red

blood cells to tens of meters for aircrafts aerodynamicity analysis. However, all the cases in this spectrum need to be correctly represented in the virtual environment at a scale which is appropriate for the user.

At the same time, it is often interesting to analyze finer details of a model, like a particularly turbulent area or a small zone of higher pressure. This means that the user should be able to enlarge and reduce the mesh accordingly.

For this purposes, a `SizeManager` has been defined and implemented. Currently, it offers the following functions:

- **Autoresize**: automatically resize the mesh to a fixed maximum length at loading time. This can be disabled in the editor, and the parameter for the maximum length can be changed.
- **Scale up**: increase the size of the mesh.
- **Scale down**: decrease the size of the mesh.

By analyzing popular implementation of zooming and scaling functions, it has been observed that the most natural solution for the user is to modify the dimensions relatively to their current values, instead of linearly. In fact, a linear resizing often feels too slow for large models and too fast for small models, and can be problematic when the size of the model approaches zero. Consequently, one can observe the implementation of the `ScaleUp` function as an example:

```
public void ScaleUp() {
  obj.transform.localScale *= 1 + scaleSpeed * Time.deltaTime;
}
```

The previous line of code provides a simple example on how time is handled in Unity. In fact, Unity exposes a static class `Time`, of which the most important field is `deltaTime`. This field returns the time in seconds it took to complete the last frame (i.e. the delay between two subsequent calls to `Update()`), as this value

depends on the computational complexity of the operations performed in the last frame and is thus not known at compile time.

3.3.8 Animation

As stated before, time-dependent datasets are a very important and interesting subset of CFD models, and the most natural way to represent them is through an animation.

For this purpose, an `AnimationManager` has been designed, and it currently offers the following functionalities:

- **Play** the animation,
- **Pause** the animation,
- **Loop** the animation.

When a time-dependent object is exported from ParaView, each frame is encoded separately as an independent model. Then, through the TCP connection between ParaView and Unity (see Section 4.3.2), a message is sent to Unity with the total number of frames to be imported. They are then read and inserted to the scene. In the current implementation, all frames are children of the same root object, and only one of them is showing at any given time. Therefore, animating the mesh consists in hiding the current frame and showing the next one. This is done in the `Update()` function that every scripts inherits from the `MonoBehavior` class, with the visibility toggling logic being encapsulated in the `ShowNextFrame()` method.

```
// Called at every frame
void Update() {
  if(isPlaying) {
    // DELAY_COUNT slows the animation
    if (delay >= DELAY_COUNT) {
      delay = 0;
      ShowNextFrame();
    } else {
      delay++;
    }
  }
}

private void ShowNextFrame() {
  // Hide all frames
  foreach (Transform child in obj.transform)
    child.gameObject.SetActive(false);

  // Show next one
  obj.transform.GetChild(currentFrame).gameObject.SetActive(true);

  // Update counter
  currentFrame = (currentFrame + 1) % obj.transform.childCount;

  // Trigger event
  if (NextFrameLoadedCallbacks != null)
    NextFrameLoadedCallbacks();
}
```

As one can notice from the last lines of the `ShowNextFrame` function, the `AnimationManager` defines a custom event called `NextFrameLoaded`, to which any other manager can bind their callbacks. This is necessary as changing the frame is basically equivalent to changing the model, and therefore some updating might be necessary in the state of other parts of the software. As seen in Section 3.3.6.1, an example of this is the recalculation of the bounding box dimensions, which must be adapted to the new shape of the model at every frame.

In the current implementation, in case of time-independent models, i.e. models that consist of only one frame, the Play/Pause button on the radial menu (see Section 3.3.5) is disabled, and therefore none of the `AnimationManager`'s routines are called.

Figure 3.9 displays four different frames of an ongoing animation. The model shown in the screenshots is taken from ParaView's official tutorial examples.

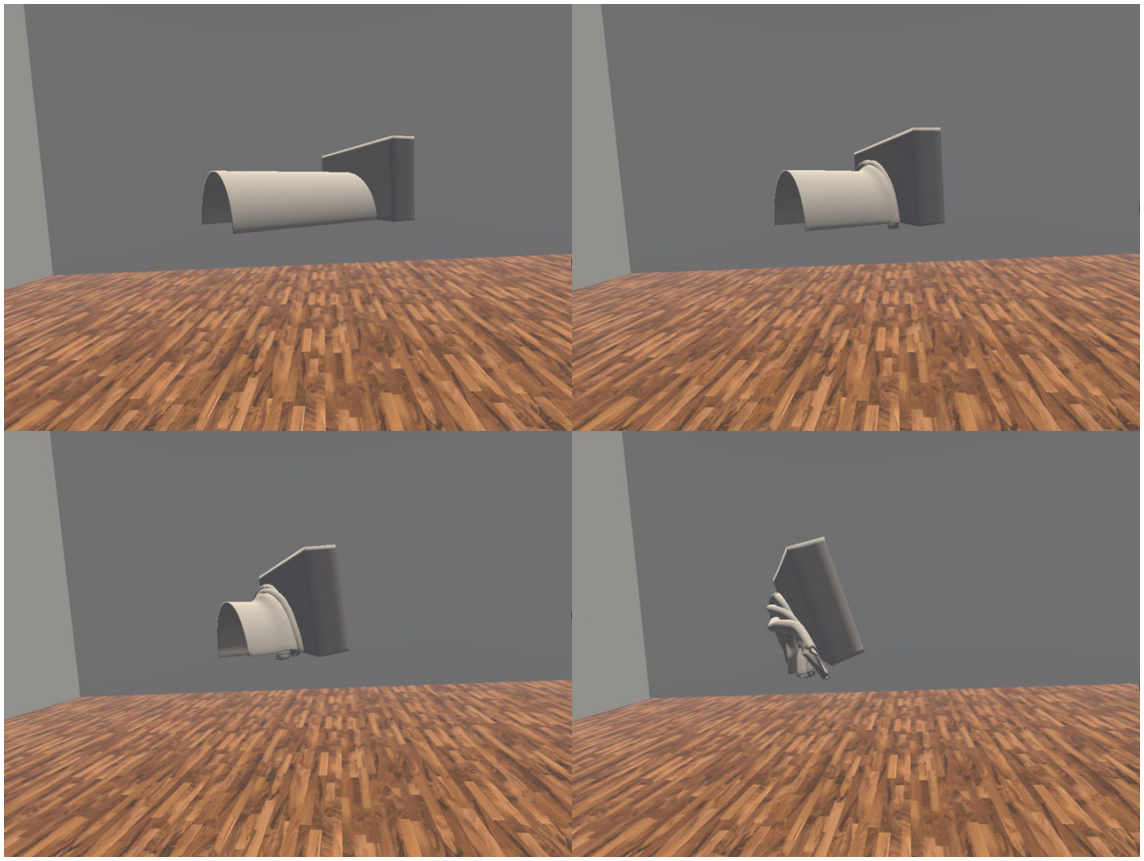

FIGURE 3.9: Screenshots of four different frames of an ongoing animation

3.3.9 Slicing

As any user of ParaView knows, the application is used not only for visualizing the data, but also to manipulate it in many ways through the use of filters. Such functions, which include *slicing*, *contouring*, *wireframing*, etc., are necessary to offer a more detailed view about the values of the dataset being analyzed. It is therefore important to be able to port these features in the Virtual Reality environment, so that this can become as independent from ParaView as possible, without requiring the user to change and re-export the data multiple times.

4.2.1 Plug-in implementation

ParaUnity is a much simpler piece of software, compared to the Unity application, and therefore the vast majority of its logic is included in just one class, called `Unity3D`. This class offers the following functionalities:

- Connecting to an existing TCP Socket and communicating through it (see Section 4.2.2).
- Spawning an instance of a pre-prepared Unity application from its executable, or connecting to a running instance of the Unity Editor.
- Requesting an X3D representation of the currently loaded objects to VTK.
- Writing data to disk.

Most of these features are implemented using Qt libraries or the Windows API.

4.2.1.1 TCP Socket

In its initial stage, ParaUnity communicated to external software through single-use TCP connections. This was done by instantiating a `QTcpSocket`, which is part of the Qt Network library and writing a byte representation of the message on it. It is worth noting that in this implementation, ParaUnity is not waiting for a reply, nor it has any methods to read it, as the TCP connection is destroyed immediately after sending the message. This can be seen in the following implementation:

```
bool Unity3D::sendMessage(const QString& message, int port) {
  // Creates the socket
  QTcpSocket *socket = new QTcpSocket(this);

  // Connects to localhost
  socket->connectToHost("127.0.0.1", port);

  // Checks connection status
  if (!socket->waitForConnected()) {
    return false;
  }

  // Writes message
  socket->write(QByteArray(message.toLatin1()));

  // Waits and disconnects
  socket->waitForBytesWritten();
  socket->waitForDisconnected();

  return true;
}
```

Connections are always done to *localhost*, although the port can vary (as described in 4.2.2).

4.2.1.2 Exporting the objects

To obtain an X3D representation of the data loaded in ParaView, ParaUnity uses a `vtkX3DExporter`, which is part of the VTK library. In fact, after instantiating the exporter, the plug-in links it to the current render window, sets the output to a file with an appropriate filename and sends the command to write the data to disk, as can be seen in the following code:

```
void Unity3D::exportScene(/* ... */) {

  /* ... */

  // Instantiate the exporter
  vtkX3DExporter *exporter = vtkX3DExporter::New();

  /* ... */

  // Set its input to the render window
  exporter->SetInput(renderProxy->GetRenderWindow());

  // Set the filename
  exporter->SetFileName(outputFilename);

  // Write to file
  exporter->Write();

  /* ... */
}
```

4.2.2 Initial communication protocol

As stated before, ParaUnity and Unity communicate through both a local TCP Socket and disk I/O operations. The actual communication protocol, as it was in the initial stage of development, can be summarized as follows:

1. Unity starts, and immediately creates a TCP Listening socket on a random port.
2. In a standardized location, Unity creates an empty directory, the name of which is the port number.
3. ParaUnity starts and searches the standardized location for files. For every file it finds (i.e. every registered port), it checks for the status of the connection on that port by sending a test message.
4. If ParaUnity finds an active Unity instance, it starts exporting the scene in the instance's directory.

5. When ParaUnity is done exporting, it sends a message to Unity with the path of the file (or the path of the directory, in case of time-dependent data).

6. When Unity receives the message, it reads the file from that location, interprets it into an object and inserts it into the scene.

4.3 Development

As explained in Section 4.2.1, the official ParaUnity plug-in suffered from performance limitations, which could hinder the user experience especially with large datasets. To overcome this problem, a custom version of ParaUnity has been designed, with particular focus on two major aspects: data I/O and inter-process communication.

4.3.1 From Disk I/O to RAM I/O

The average mechanical hard drive can perform around 300 I/O random-access operations per second. With solid-state drives, such value can go up to about 2000. However, this is still several order of magnitude lower than the speed of an average RAM, which can perform about 40 million random access I/Os every second [26].

With this numbers in mind, it becomes clear that one of the performance bottlenecks of the initial implementation of ParaUnity is disk I/O.

To alleviate this issue, and take advantage of the much-higher speed offered by main memory, it has been decided to move the transmission of the data to RAM. On a Windows machine, one of the possible solutions (and the one adopted for the project) is to interface with the Windows API and store the data as *Named Shared Memory objects* [27]. The work-flow can be summarized as follows:

1. Process A creates a *file mapping* in the main memory of the system, i.e. requests the Operating System to allocate a specific amount of memory and give it a custom name.

2. Process A maps part of the memory in its process space with the memory requested in the previous step, so that they can be effectively treated as the same part of memory.

3. Process A writes the data to the portion of its space which is mapped to shared memory; consequently, the data becomes accessible to any process through the name set in Step 1.

4. Process B, which must know the name and the size of the shared object beforehand, accesses the file mapping in shared memory by name.

5. Process B maps the area of the shared memory in which the object is written to a portion of memory in the process space.

6. Process B can now access the object as if it were in its own memory space.

7. When both processes are done accessing the files, Process A frees the memory in shared memory and they both unmap their view of the file.

This has been implemented in ParaUnity by setting the `vtkX3DExporter` to return a string of data instead of writing to a file, and copying that string to shared memory with the following logic: