MagicVR - 3D Models and Effects

Jerome D. Pnisch* 3D Models and Effects

Michael Maier† Gesture Recognition

ABSTRACT

This documentation is about MagicVR - a virtual reality application developed in OpenSG for the LRZ V2C in Garching supporting 3D-Gesture-Recognition and Lerp-Animations.

Index Terms: V2C—Virtual Reality—OpenSG—Treemaps

1 INTRODUCTION

In this documentation the concept and implementation of MagicVR will be explained. MagicVR is a virtual reality application which was developed in c++ and OpenSG in order to run in the V2C mixed reality cave of the LRZ in Garching, Munich. It's main feature is a 3D-Gesture-Recognition opening a whole world of interaction possibilities using only the wand (or any tracker) in a 3D tracking system. A lot effort also was put in the front-end of MagicVR using FractTime-Animations to easely define and control effects and movements within the virtual reality scene.

2 CONCEPT

MagicVR was developed in a cooperation between Jerome Pnisch and Michael Maier. This paper will be focused on the 3D Art and FractTime-Animation effects done by Jerome Pnisch while Michaels's paper will be focused on the 3D-Gesture-Recognition. This section shall introduce in the general concept of MagicVR and the effects and animation part.

2.1 The idea behind MagicVR

The main idea of MagicVR is introducing the 3D-Gesture-Recognition module in a playful environment using important features of a virtual environment. A user/player should be animated to use 3D-Gestures as interesting interaction technique rather than pushing buttons. The virtual environment should help to gain the user/player's interest and make him feel comfortable while experiencing the mixed reality world. As User-Feedback it was decided to focus on animations and movements within the scene.

2.2 FractTime-Animations

In order to make movements easier to implement and control a framework was implemented based on the idea of Vector Lerp, FractTime and Coroutines from Unity and C#.

Lerp is a commonly used short term standing for "linear interpolation". The concept of a Vector.Lerp in Unity and C# takes an original vector and a target vector and makes a linear interpolation between those two vectors using an interpolation factor from 0 to 1.

FracTime is used as the interpolation factor in our framework. It devides the already animated time by the target duration of the animation providing a factor between 0 and 1.

Coroutines are used in Unity to run multible routines at the same time which makes it easier to control them individually. The difference to c++ is, that multithreading is more trivial in C#.

Those two concepts were used in combination to write our own FractTime-Animations framework.

2.3 Advantages of FracTime Animations

The advantages of using our animation framework are among others:

Trustability Destination and Target values of vectors, colors etc. are clear defined and thereby no "surprises will happen during animations"

Controlability Since all animations are placed and controlled in one main container and subcontainers, each animation can be started and stopped individually.

FPS-Stability Since the animations are based on the FracTime, which is based on the delta time between frames, and not e.g. the translation distance animations will allways take the same time independent of the FPS given by the render system.

3 IMPLEMENTATION

In this section some implementation details will be given for the animations and effects.

3.1 Environment

An important part in the user experience with mixed and virtual reality plays the virtual environment. Here a combination of a skybox and an environment 3D-model was used to get the most out of the V2C cave regarding to immersion. The skybox image was rendered using an HDRI image and a self written hdri to image converter in Blender 3D.

Figure 1: Stonehenge by ruslans3d is licensed under CC Attribution https://sketchfab.com/models/37cfc2bb99944703b5d57ea281030ca6

*e-mail: jerome.poenisch@campus.lmu.de

†e-mail: micahel.maier@campus.lmu.de

Figure 2: Source: https://hdrmaps.com

Listing 1: Implementing the Skybox

```
SkyBackgroundUnrecPtr loadBackground(
    Resolutions skyboxResolution) {

 std::string bgImageFilePath =
     Path_to_Skybox_image;

 ImageUnrecPtr mainImage = ImageFileHandler
     ::the()->read(bgImageFilePath.c_str());

 /**
 * Create empty images as destinations for
     subimage */
 ImageUnrecPtr imgFront = ImageBase::create
     ();
 ImageUnrecPtr imgBack = ImageBase::create()
     ;
 ImageUnrecPtr imgLeft = ImageBase::create()
     ;
 ImageUnrecPtr imgRight = ImageBase::create
     ();
 ImageUnrecPtr imgTop = ImageBase::create();
 ImageUnrecPtr imgBottom = ImageBase::create
     ();

 /**
  * Easier to change Skybox Images by
      croping them out
  * at runtime directly from the CubeMap
      Image exported from Blender
  *
  * Blender exports the environment maps in
      the format:
  *
  * Left   | Back | Right
  * Bottom | Top  | Front
  *
  * While subImage reads
  * - offX from left to right
  * - offY from bottom to top */
 mainImage->subImage(2 * res, 0  , 0,
   res, res, 1,
   imgFront);
 mainImage->subImage(res , res, 0,
   res, res, 1,
    imgBack);
 mainImage->subImage(0  , res, 0,
   res, res, 1,
   imgLeft);
 mainImage->subImage(2 * res, res,
   0, res, res, 1,
   imgRight);
 mainImage->subImage(res, 0  , 0,
   res, res, 1,
   imgTop);
 mainImage->subImage(0 , 0  , 0,
   res, res, 1,
   imgBottom);

    /**
     * Create Textures and set image to
         cropped images */
    TextureObjChunkUnrecPtr texFront =
        TextureObjChunk::create();
    texFront->setImage(imgFront);

    TextureObjChunkUnrecPtr texBack =
        TextureObjChunk::create();
    texBack->setImage(imgBack);

    TextureObjChunkUnrecPtr texLeft =
        TextureObjChunk::create();
    texLeft->setImage(imgLeft);

    TextureObjChunkUnrecPtr texRight =
        TextureObjChunk::create();
    texRight->setImage(imgRight);

    TextureObjChunkUnrecPtr texTop =
        TextureObjChunk::create();
    texTop->setImage(imgTop);

    TextureObjChunkUnrecPtr texBottom =
        TextureObjChunk::create();
    texBottom->setImage(imgBottom);

    /**
     * Create Skybox and set textures */
    SkyBackgroundUnrecPtr skyBG =
        SkyBackground::create();
    skyBG->setFrontTexture(texFront);
    skyBG->setBackTexture(texBack);
    skyBG->setLeftTexture(texLeft);
    skyBG->setRightTexture(texRight);
    skyBG->setTopTexture(texTop);
    skyBG->setBottomTexture(texBottom);

    return skyBG;
}
```

3.2 Animations

While the *Lerp* and the *FracTime* was easy to implement in c++, for the *Coroutines* had to be found another solution. To make things easy an animation holder was implemented, controlling all animations of the scene in one place.

```
void Scene::update(OSG::Time dTime) {
 _animations.animate(dTime);
}
```

The framework which was implemented uses two main animation classes “ParallelAnimation” and “SequentialAnimation” defining as the name lets guess already whether contained animations shall be performed simultaneously or sequentially. So it’s easy to guess

that the Scene::_animations is a ParallelAnimation animating all contained animations at the same time.

Listing 2: ParallelAnimation::animate

```
void ParallelAnimation::animate(OSG::Time
    dTime) {
 for (auto animation : _animations) {
  animation->animate(dTime);
 }
 removeStoppedAnimations();
 if (_stopIfNoAnimations && _animations.
     empty()) {
  stop();
 }
}
```

Listing 3: SequentialAnimation::animate

```
void SequentialAnimation::animate(OSG::Time
    dTime) {
 if (_stopIfNoAnimations && _animations.
     empty()) {
  stop();
 } else {
  _animations.front()->animate(dTime);
  if (_animations.front()->isStopped()) {
   _animations.pop();
   if (_stopIfNoAnimations && _animations.
       empty()) {
    stop();
   }
  }
 }
}
```

Then there are some animation wrappers like “EaseInAnimation”, “EaseOutAnimation” and “BezierAnimation” allowing to implement completely ease controlled animations and even a forward-backwards animation as seen in the example below:

Listing 4: BezierAnimation::animate

```
void BezierAnimation::animateFracTime(OSG::
    Time fracTime) {
 _animation->animate(_bezier.atPercentage(
     fracTime).y());
}
```

Finally there are the actual animation performing the before mentioned Lerp on Vectors “TranslationAnimation” and “ScaleAnimation” as seen in the example below:

Listing 5: TranslattionAnimation::animate

```
void TranslationAnimation::animateFracTime(
    OSG::Time fracTime) {
 Vec3f _movement = _destination - _start;
 _trans->setTranslation(_start + _movement *
      fracTime);
}
```

3.3 Effects

Based on this animations framework the effects and user feedback was implemented using for example a TranslationAnimation and a ScaleAnimation wrapped in a ParallelAnimation with repetition enabled:

Listing 6: BubbleAnimationsNode::animate

```
void BubbleAnimationsNode::addAnimations() {
 for (const auto &data : _bubbleDatas) {
  _animations.add(std::shared_ptr<Animation
      >(
   new ParallelAnimation{
    std::shared_ptr<Animation>(
     new TranslationAnimation(
      data.transformation,
      OSG::Vec3f(0, 1, 0),
      3 * data.data, true)),

    std::shared_ptr<Animation>(
     new ScaleAnimation(
      data.transformation,
      OSG::Vec3f(0, 0, 0),
      3 * data.data, true))
     }
  ));
 }
}
```

4 Use cases

In this section we come to some use cases of the FracTime animations, the user gameplay and instruction of MagicVR.

TODO! pictures

4.1 Element Stones

In the virtual scene are placed some columns with “Elementary Stones”. They have different symbols as texture representing the according element and at the same time the 3D-Gesture which has to be performed to call this element. Those “Elementary Stones” can also be used to practice and get used to the 3D-Gestures.

If the a 3D-Gestures of one of the elements is correctly recognized, a BubbleAnimation as explained before starts to run, letting the user know, which element he activated.

4.2 Shooting Elements

If practiced enough, the user can feel free to move on to shoot elements. This is done by a combination of 3D-Gestures:

- 1. Make a circle gesture: This activates the “Shoot Mode”. You will see this as there should appear a neutral ball at the top of your wand.
- 2. Make the element gesture of your wish: This calls the element. You should see that the ball at the top of your wand changes into the according element - equal to the element bubbles you saw on the elementary stones.
- 3. Make a multiplication gesture: This allows later on to shoot multiple element bubbles at once according to how many times you use this gesture. You should notice, that the ball at the top of the wand grows according to the multiplication.
- 4. Shoot the element using the shoot gesture. The element bubbles will shoot out of your wand searching its way to the according interaction. If you had multiplication done, you will see the ball shrink while shooting multiple bubbles.

4.2.1 Fire and Water

TODO! pictures

Element bubbles of the type water and fire will have their target on the fireplace located at the left side of the scene. Fire bubbles make the fire grow bigger while water will shrink it down.

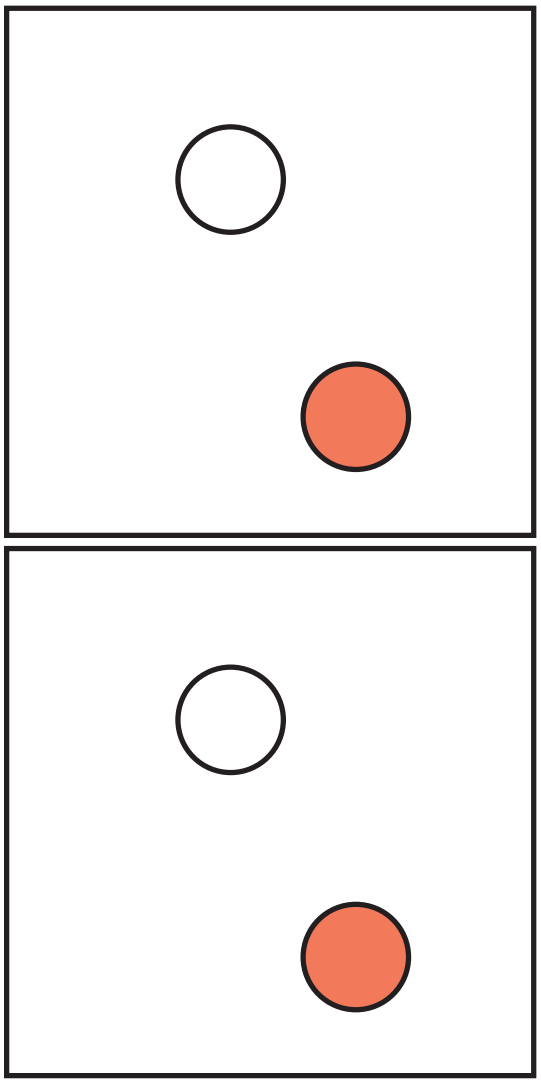

Figure 3: Scene Overview

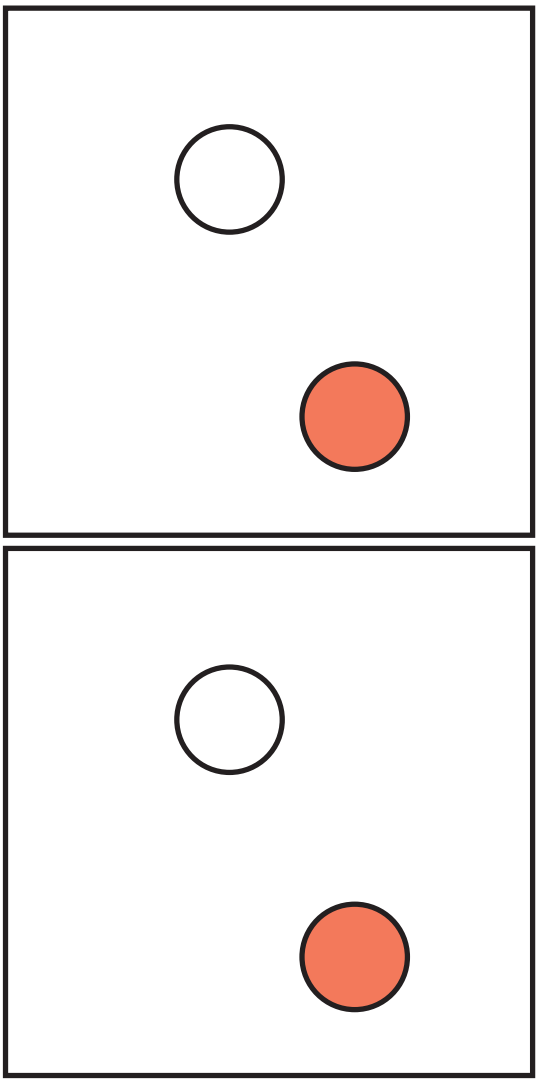

Figure 4: Fire and Water gestures

4.2.2 Light

TODO! picture

Element bubbles of the type Light will have their target at the lantern placed in the front part of the scene. As it is hit by the bubbles, the scene will be lit more and more.

4.2.3 Wind

TODO! picture
TODO! description

5 Outlook

The implemented animation framework makes it easy to implement further animation types and more complex movements while they stay very control- and adjustable. There are multiple further ways how and where this animation framework could be applied e.g. animating not only Vectors but also materials, colors, transparency etc.

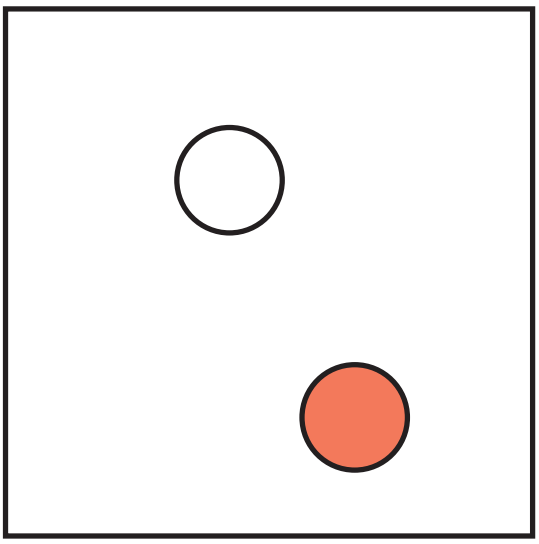

Figure 5: Light gesture

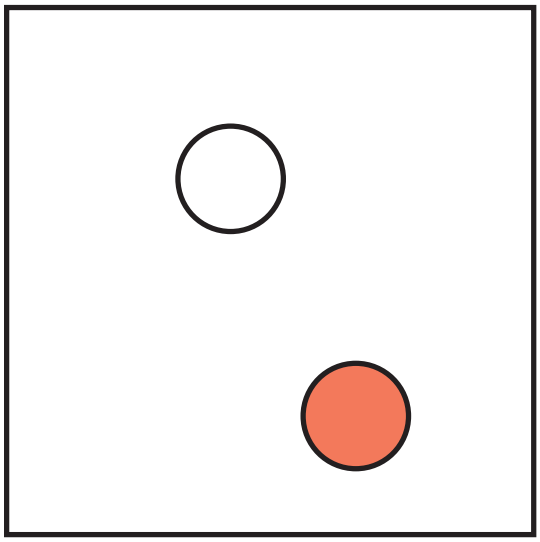

Figure 6: Wind gesture