UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Whoosh Division

OcuViz - EpiUse Labs

Software Requirements Specification
and Technology Neutral Process Design

Vukile Langa	u14035449
Wynand Hugo Meiring	u13230795
Nontokozo Hlastwayo	u14414555
Gerome Schutte	u12031519

October 19, 2016

Contents

1 Vision

OcuViz is a platform for creating rich visual representations of otherwise unintuitive data by using the power of virtual reality visualisation. It aims to address the time consuming task of working with 3D scenes and make it simple, by providing a format for specifying scenes that is concise, optimised for integration with pluggable data, and to be targeted at being easily usable by anyone with even the slightest development experience.

OcuViz would typically be deployed in research, data-auditing and demonstration environments, where visualisations may be used as a medium to bring data to life, for example, describing a scene that gives viewers an interactive walk-around view of a solar cataclysm rather than simply listing numbers representing objects and scale, or allowing viewers to encounter the world from the point of view of a tiny animal, or giving demonstrators the power to transport an audience to a previously unseen location.

2 Background

EPI-Use Labs provides products aimed at getting the most out of data. The business prides itself in deploying products which allow users to sync, manipulate, extract and report various forms of data, whether through on-site or cloud-based solutions. The OcuViz project is a natural evolution of this focus. OcuViz enriches the data-reporting process for in-house developers at EPI-Use Labs and business clients alike, by adding new visual and interactive dimensions throughout their project life-cycles, whether used in product development as a tool to interact with test and debugging data, or as a value added feature in a complete solution.

3 Architecture Requirements

3.1 Access Channel Requirements

The system should be accessible via a desktop client, with mobile clients specified as optional, but most likely not feasible due to the hardware requirements of graphics processing. The target platform is Windows x64, and as such a Windows x64 client should be provided in the form of a .exe file, complete with .exe install file.

3.2 Quality Requirements

3.2.1 Performance

Stringent performance requirements are necessary, not only for novelty purposes, but due to the introduction of virtual reality. In order for visualisations running in virtual reality not to cause motion sickness in the viewer, the following minimum requirements need to be met:

- Visualisations produced must run at a constant framerate of 75fps or more.
- Overall system latency between user input and display must be kept below a maximum of 20ms.

3.2.2 Reliability

- Since input data and scene descriptor files may optionally in large part be user- or externally generated, faulty input data and scene descriptor files must be detected and reported without system crash.
- All expected user inputs for each visualisation must be clearly defined. Where an input is not declared as "expected", it must have no effect.
- The results of rendering a visualisation from a scene descriptor and input data file must be predictable and repeatable.
- Objects in a scene created via the scene editor must have the same properties in the actual visualisation as specified the scene editor.
- All visible objects specified in the scene descriptor file or placed via the scene editor must be rendered in the visualisation.

3.2.3 Scalability

- The system must be developed using technologies which are operating system neutral. Windows x64 is specified as the priority target platform, but the client may in future choose to port the system to another operating system, and must be able to do so without having to rewrite system components.

3.2.4 Flexibility

- Visualisations must be renderable using either "hard-coded" objects fully specified in a scene descriptor file, or object blueprints specified in a scene descriptor file with "variable" object properties that may be read from an input data file.
- Input data files must be pluggable, provided they contain valid input data as required by the object blueprints listed in the scene descriptor file.

3.2.5 Maintainability

- Version control must be used throughout the development process to create a centralised point auditing and review of code changes and simple rolling back to working system versions.
- System components must be designed to be modular, following a separation of concerns approach, in order to ease pin-pointing of system errors and swapping out of existing system modules.

3.2.6 Cost

- The client has not budgeted for additional technologies or hardware, and as such, all technologies used in development must be free for use.

3.2.7 Usability

- A complete manual must be provided describing the valid format and possible inputs of a scene descriptor file.
- Only two creation contexts for visualisations are specified: importing a scene descriptor and optional input data files, and using the scene editor. Within these contexts, each use case must have a single point of access to contribute to use cases being intuitive and predictable.
- A running visualisation must have clearly visible access to visualisation variable controls, such as viewer scale and position.

3.3 Architecture Constraints

3.3.1 Required Technologies

- **Software:**
  - Microsoft Windows 7 SP1 or newer x64 version operating system.
- **Hardware:**
  - Oculus Rift Virtual Reality Headset
  - Due to the processing requirements of virtual reality and the Oculus Rift, a host machine is required with:
    - Graphics: NVIDIA GTX 970 / AMD 290 equivalent or greater
    - Processor: Intel i5-4590 equivalent or greater
    - 8GB+ RAM
    - HDMI 1.3 video output port
    - 2x USB 3.0 ports

3.3.2 Architectural Strategies

- **Performance Strategies:**
  In order to meet the demands of graphics processing, the framework used must support
  - Resource-reuse using caching of objects which move in and out of the scene
  - Object pooling for duplicating repeated objects in the scene which have already been processed.

  If the framework used doesn't provide native support for the abovementioned, it must support ways in which a custom implementation may be created. Furthermore, if processing still doesn't meet the set performance requirements, an increase in hardware processing power will be required.
- **Reliability Strategies:**
  To be able to develop a system with reliable and predictable behaviour, the framework used must support
  - Fault detection mechanisms
  - Exception communication to be used throughout the rendering process, specifically in the event of invalid input
  - Self-testing service providers which ensure the validity of input arguments and generated output.

  System faults must be prevented as much as is reasonably possible using
  - A test-driven development approach

    - A testing framework which integrates with the framework used in developing the system.

- **Flexibility Strategies:**
  Multiple forms of flexibility strategies must be employed:
    - Process Flexibility:
        * A dedicated workflow controller must be responsible for overseeing the entire object parsing process.
        * Each system module must employ responsibility localization.
    - Service Provider Flexibility:
        * System modules and service providers must be contracts based.
        * Objects of various predefined types must be created using abstract factories.
        * Dependency injection must be employed across the system.
    - Flexibility Support:
        * Automated builds must be run on each code repository change using a continuous integration server.
        * Automated testing of all use cases must be run on each automated build.

4 Functional Requirements and Application Design

4.1 Use Cases

1. **Add Object**
   Adding an object to the scene by selecting a shape from the creation editor-section and clicking the position in the scene where it should be placed.
    - **Priority:** Critical
    - **Service contract:**
        - **Post-conditions:**
            * Object is created
            * Object has default properties associated with the selected shape
            * Object is placed in scene at selected position.
            * Object is added to an object pool of its type.
2. **Edit Object**
   Selecting an object in the scene object hierarchy and viewing and possibly editing its properties such as scale, texture, surface colour, etc. via the property inspector editor-section.
    - **Priority:** Critical
    - **Service contract:**

- Pre-conditions:
  - The selected object exists in the scene.
- **Post-conditions:**
  - Object's properties are updated to reflect changes made.
  - Object's visual representation in scene editor reflects changed properties.

3. **Wrap Object in Collection**
   Selecting an object in the scene object hierarchy and wrapping it in a collection via the collection selection menu in the creation editor-section. The collection becomes recognized as an object itself, parent to the object it wraps in the context of the scene object hierarchy.

   - **Priority:** Critical
   - **Service contract:**
     - **Pre-conditions:**
       - The selected object exists in the scene.
     - **Post-conditions:**
       - A collection of duplicate objects is visualised in the scene.
       - The collection object is created.
       - The collection is set as the subject object's parent in the scene object hierarchy.

4. **Move Object**
   Selecting an object in the scene object hierarchy and repositioning it via mouse drag.

   - **Priority:** Important
   - **Service contract:**
     - **Pre-conditions:**
       - The object exists in the scene.
     - **Post-conditions:**
       - The object's position properties are updated to reflect changes made.
       - Object's visual representation in scene editor reflects changed position.

5. **Select Input Data**
   Indicating a new input data file via the creation editor-section and specifying its location via a directory open dialog.

   - **Priority:** Important
   - **Service contract:**
     - **Post-conditions:**
       - The input data file object is created.

6. **Bind Input Data**
   In the same manner as use case 2, edit an object's properties to indicate, for each individual bound property, that it is bound and to which input data variable to bind it.

   - **Priority:** Critical
   - **Service contract:**
     - **Pre-conditions:**
       * The object exists in the scene.
       * The input file exists in the scene.
       * The input file contains column names matching variable names supplied by the user.
     - **Post-conditions:**
       * The object's bound property values are updated to match the input data values.
       * If more than one row of data is available for the input data supplied, duplicate objects are created for each row. Each duplicate object's bound property value corresponds to the row value of the input data.

7. **Export Scene**
   Exporting an editor-created scene to a .csv file that may be read, edited, and rendered at a later stage by importing it.

   - **Priority:** Nice-to-have
   - **Service contract:**
     - **Pre-conditions:**
       * No scene errors must exist.
     - **Post-conditions:**
       * A .csv file must be created in the location specified, containing a scene descriptor that may be used to reproduce the scene that was exported.

8. **Import Scene**
   Importing a user- or editor-created .csv file to be processed and rendered.

   - **Priority:** Critical
   - **Service contract:**
     - **Pre-conditions:**
       * The .csv file must exist.
       * The .csv file format must conform to the scene descriptor format laid out in the user manual.
     - **Post-conditions:**
       * A scene is created.
       * The scene object hierarchy is populated and directly corresponds to the objects mentioned in the scene descriptor .csv file.

        * Objects created have properties which directly correspond to the objects mentioned in the scene descriptor .csv file.

9. **Parse Instruction**
   Processing a line of input from the scene descriptor .csv file into an object which may be rendered in the scene.

   - **Priority:** Critical
   - **Service contract:**
     - **Pre-conditions:**
       * The line is non-empty.
       * The line's format is error-free, conforming to the scene descriptor format laid out in the user manual.
     - **Post-conditions:**
       * An object is created with properties based on the data values in the line.

10. **Search Model**
    Perform a remote search for custom object models (in .obj format) which may be downloaded and included in a scene.

    - **Priority:** Nice-to-have
    - **Service contract:**
      - **Pre-conditions:**
        * The search query must be non-empty.
      - **Post-conditions:**
        * A resulting set of models are returned from the remote model store which correspond to the search query, or a message is displayed that an error occurred in case of network error.

11. **Import Model**
    Select a custom object model from a remote model store and import it, downloading it and making it possible to be included in the scene.

    - **Priority:** Nice-to-have
    - **Service contract:**
      - **Pre-conditions:**
        * The model selected must be freely available.
      - **Post-conditions:**
        * The model's .obj file is downloaded to the scene's model folder.

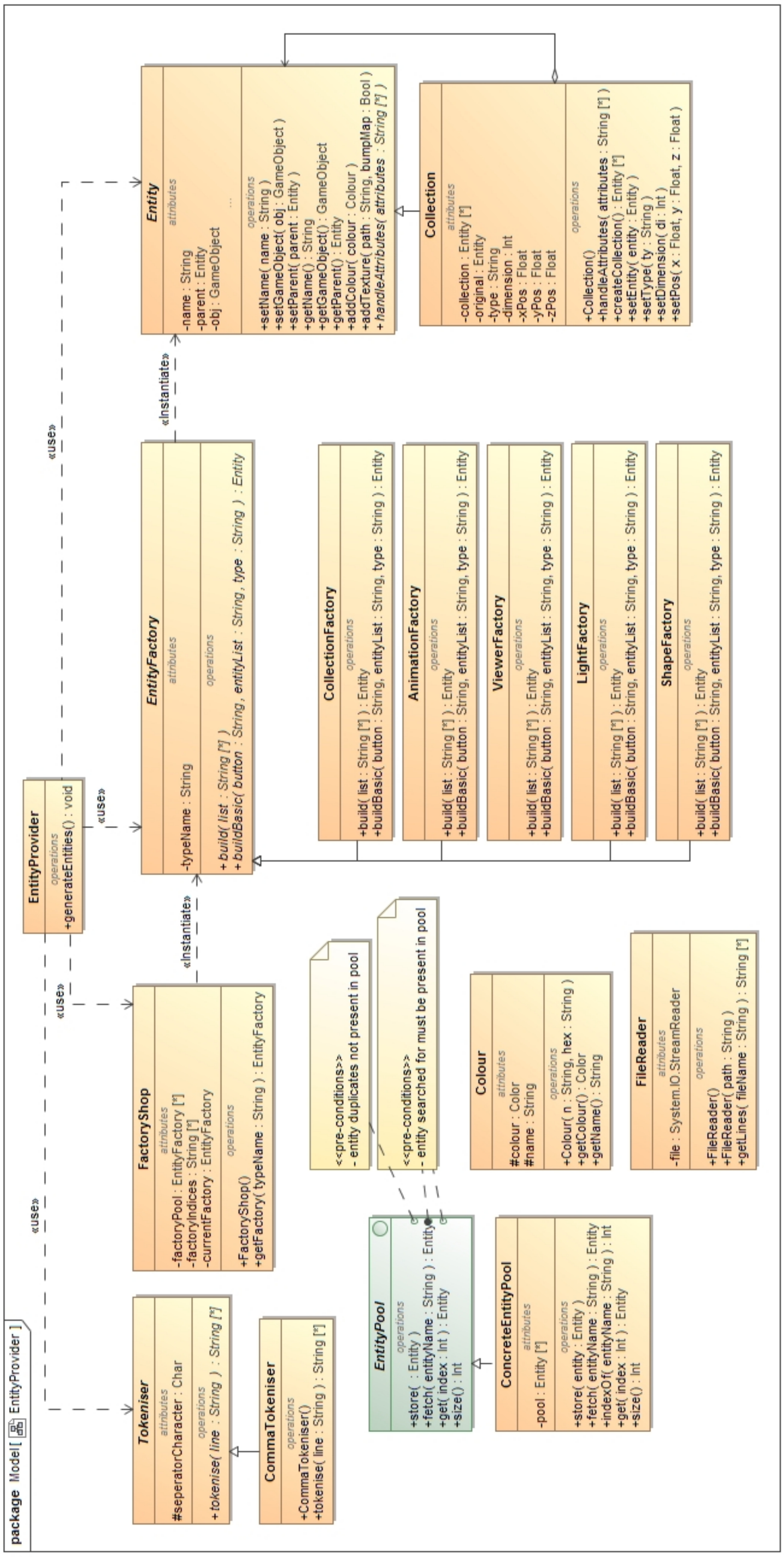
12. **User Controls via LeapMotion**
    Using the LeapMotion controller, allow users to control aspects of a visualisation, such as movement around a scene or adjusting scale.

    - **Priority:** Nice-to-have

- **Service contract:**
  - **Pre-conditions:**
    - The input type is defined.
  - **Post-conditions:**
    - The visualisation reflects changes depending on what the user input is defined to do.

4.2 Domain Model

The domain model for the EntityProvider package, which parses and renders objects, follows.


package Model[EntityProvider]
EntityProvider
operations
+generateEntities() : void
«use»
«use»
«use»
«use»
«Instantiate»
«Instantiate»
Tokeniser
attributes
#seperatorCharacter : Char
operations
+ tokenise(line : String) : String [*]
CommaTokeniser
operations
+CommaTokeniser()
+tokenise(line : String) : String [*]
FactoryShop
attributes
-factoryPool : EntityFactory [*]
-factoryIndices : String [*]
-currentFactory : EntityFactory
operations
+FactoryShop()
+getFactory(typeName : String) : EntityFactory
EntityFactory
attributes
-typeName : String
operations
+ build(list : String [*]) : Entity
+ buildBasic(button : String, entityList : String, type : String) : Entity
Entity
attributes
-name : String
-parent : Entity
-obj : GameObject
...
operations
+setName(name : String)
+setGameObject(obj : GameObject)
+setParent(parent : Entity)
+getName() : String
+getGameObject() : GameObject
+getParent() : Entity
+addColour(colour : Colour)
+addTexture(path : String, bumpMap : Bool)
+ handleAttributes(attributes : String [*])
Collection
attributes
-collection : Entity [*]
-original : Entity
-type : String
-dimension : Int
-xPos : Float
-yPos : Float
-zPos : Float
operations
+Collection()
+handleAttributes(attributes : String [*])
+createCollection() : Entity [*]
+setEntity(entity : Entity)
+setType(ty : String)
+setDimension(di : Int)
+setPos(x : Float, y : Float, z : Float)
CollectionFactory
operations
+build(list : String [*]) : Entity
+buildBasic(button : String, entityList : String, type : String) : Entity
AnimationFactory
operations
+build(list : String [*]) : Entity
+buildBasic(button : String, entityList : String, type : String) : Entity
ViewerFactory
operations
+build(list : String [*]) : Entity
+buildBasic(button : String, entityList : String, type : String) : Entity
LightFactory
operations
+build(list : String [*]) : Entity
+buildBasic(button : String, entityList : String, type : String) : Entity
ShapeFactory
operations
+build(list : String [*]) : Entity
+buildBasic(button : String, entityList : String, type : String) : Entity
<<pre-conditions>>
- entity duplicates not present in pool
<<pre-conditions>>
- entity searched for must be present in pool
EntityPool
operations
+store(: Entity)
+fetch(entityName : String) : Entity
+get(index : Int) : Entity
+size() : Int
ConcreteEntityPool
attributes
-pool : Entity [*]
operations
+store(entity : Entity)
+fetch(entityName : String) : Entity
+indexOf(entityName : String) : Int
+get(index : Int) : Entity
+size() : Int
Colour
attributes
#colour : Color
#name : String
operations
+Colour(n : String, hex : String)
+getColour() : Color
+getName() : String
FileReader
attributes
-file : System.IO.StreamReader
operations
+FileReader()
+FileReader(path : String)
+getLines(fileName : String) : String [*]

5 Open Issues

The following issues are left unaddressed by this document:

- No request and result data structures are indicated for use cases.
- Required functionality is not indicated using use case diagrams.