**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

**ДИПЛОМНА РАБОТА**

Тема: 3D pедактор за мобилна платформа Andriod.

Дипломант: Научен ръководител:

Антонио МилевКирил Митов

СОФИЯ

2019

**TЕХНОЛОГИЧНО УЧИЛИЩЕ**



**ЕЛЕКТРОННИ СИСТЕМИ**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

Дата на заданието: 06.11.2018г. Утвърждавам:..............................

Дата на предаване: 06.02.2019г. /проф. д-р инж. Т. Василева/

**ЗАДАНИЕ**

**за дипломна работа**

на ученика Антонио Ерол Милев 12 Б клас

1.Тема: 3D pедактор за мобилна платформа Andriod.

2.Изисквания:

2.1 Възможност за създаване и редакция на основни 3D обекти

2.2 Възможност за вмъкване и съхранение от и към основни 3D формати.

3.Съдържание 3.1 Обзор

3.2 Същинска част

3.3 Приложение

Дипломант :...........................................

Ръководител:..........................................

/Кирил Митов/

Директор:................................................

/ доц. д-р инж.Ст. Стефанова /

# Съдържание

[Съдържание 4](#_Toc2191167)

[Увод 6](#_Toc2191168)

[Първа Глава 7](#_Toc2191169)

[Преглед на използвани методи и технологии. Проучване на подобни приложения 7](#_Toc2191170)

[1.1 Съществуващи приложения 7](#_Toc2191171)

[1.1.1 3DC.io 7](#_Toc2191172)

[1.1.2 Spacedraw 7](#_Toc2191173)

[1.2 Технологии и среди за развой 8](#_Toc2191174)

[1.2.1 Intellij IDEA 8](#_Toc2191175)

[1.2.2 Android Studio 8](#_Toc2191176)

[1.2.3 Java 9](#_Toc2191177)

[1.2.4 Kotlin 9](#_Toc2191178)

[Втора Глава 10](#_Toc2191179)

[Функционални изисквания към приложението. Проектиране на редактор на 3D графика 10](#_Toc2191180)

[2.1 Функционални изисквания към 3D редактор на графика 10](#_Toc2191181)

[2.2 Избор на програмен език и библиотеки 10](#_Toc2191182)

[2.2.1 IDE - Android Studio 10](#_Toc2191183)

[2.2.2 ViroCore 10](#_Toc2191184)

[2.2.3 Език за програмиране - Java 11](#_Toc2191185)

[2.3 Проектиранe 11](#_Toc2191186)

[2.3.1 Структура на приложението 11](#_Toc2191187)

[2.3.2 Описание на фрагментите 13](#_Toc2191188)

[2.3.3 Работа с ViroCore 14](#_Toc2191189)

[Трета Глава 16](#_Toc2191190)

[Програмна реализация на редактор на 3D графика за Android 16](#_Toc2191191)

[3.1 Изграждане на първоначалната сцена 16](#_Toc2191192)

[3.2 Обработване на входни събития от фрагментите. 17](#_Toc2191193)

[3.3 Undo/Redo 18](#_Toc2191194)

[3.4 Създаване и изтриване на примитивни обекти 20](#_Toc2191195)

[3.5 Основни параметри на обекта 23](#_Toc2191196)

[3.6 Селектиране на обект 25](#_Toc2191197)

[3.7 Помощни стрелки при селектиране 25](#_Toc2191198)

[3.8 Йерархия 28](#_Toc2191199)

[3.9 Промяна на материала на обекта 32](#_Toc2191200)

[3.10 Орбитираща камера 35](#_Toc2191201)

[3.11 Вмъкване и съхранение на обекти 37](#_Toc2191202)

[Четвърта Глава 41](#_Toc2191203)

[Изисквания. Ръководство на потребителя 41](#_Toc2191204)

[4.1 Изисквания 41](#_Toc2191205)

[4.2 Инструкции за ползване 41](#_Toc2191206)

[4.2.1 Главен изглед 41](#_Toc2191207)

[4.2.1.1 Контрол на камерата 42](#_Toc2191208)

[4.2.1.2 Селектиране и манипулиране на обекти 42](#_Toc2191209)

[4.2.2 Меню за създаване на примитивни обекти 43](#_Toc2191210)

[4.2.3 Меню за промяна на параметрите на обекта 44](#_Toc2191211)

[4.2.4 Изглед на йерархията 44](#_Toc2191212)

[4.2.5 Меню за промяна на материала на обект 46](#_Toc2191213)

[4.2.6 Горно меню 48](#_Toc2191214)

[Заключение 50](#_Toc2191215)

[Използвана литература 51](#_Toc2191216)

# Увод

В днешно време много хора работят в сферата на 3D графиката. Основните инструменти, които използват са сложни софтуери за обработка като Blender, Maya, 3ds Max и много други. Съществуват, обаче, съвсем малко подобни софтуери за мобилни устройства. Разбира се, продукти с мащаби като тези на изброените трудно биха били постижими, така че да работят на повечето телефони, таблети и т.н. Тези устройства обаче предлагат повече удобство благодарение на преносимостта си. Понякога на човек може да му потрябва да промени нещо малко в някоя сцена, или да моделира набързо прост модел. В такъв случай, мобилно приложение с по-ограничени възможности за 3D обработка би му свършила работа.

Настоящата дипломна работа има за цел да се разработи точно такова приложение. То ще работи върху Android мобилно платформа и ще предоставя основен набор от функционалности за редактиране на 3D графика. Това включва възможност за създаване и редакция на основни 3D обекти и възможност за вмъкване и съхранение на моделите. Това ще позволява на потребителят да нанася промени върху преди това създадени модели както и да създава нови.

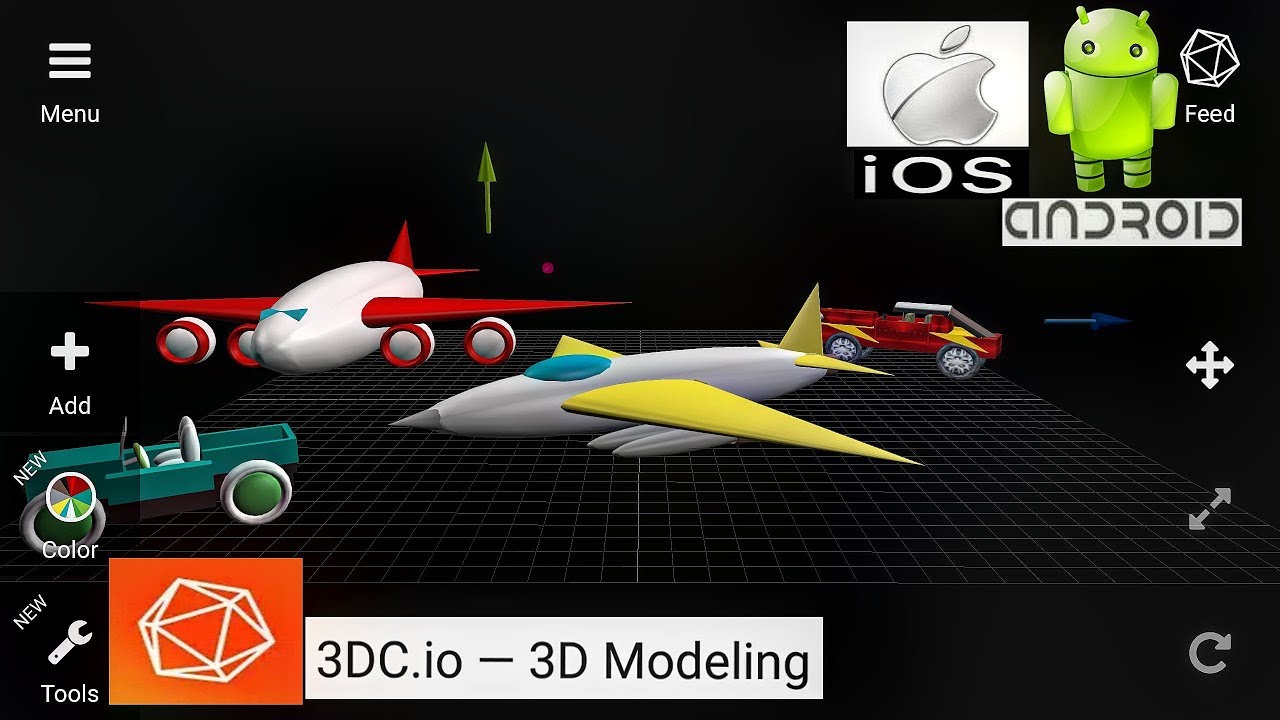
# Първа Глава

# Преглед на използвани методи и технологии. Проучване на подобни приложения

## Съществуващи приложения

### ****3DC.io****

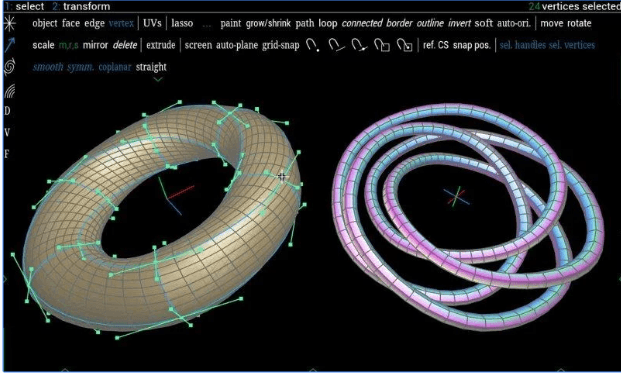
3DC.io е програма за разработка на 3D модели с много прост и интуитивен интерфейс. Има десктоп версия, както и мобилна за Android и iOS. Подходящо е за начинаещи и деца, които искат да се занимават с дизайн. Приложението има фокус върху образованието като предлага отстъпки на училища за платената си версия с цел да бъде интегрирано в учебния процес. Чрез 3DC.io потребителят може да съхранява моделите си във .OBJ и .STL формати. Липсват някой основни функционалности като връщане назад и напред.



Фигура 1.1.1.1

### Spacedraw

Spacedraw прилича повече на сложна програма за 3D обработка. Хора, които не са се занимавали преди в тази сфера би им се сторила неразбираема. За тези, които могат да го използват, Spacedraw предлага множество функционалности като работа с криви, текстури, рисуване върху повърхността на модела и др. Има безплатна и платена версия като при безплатната потребителите не могат да работят с модели надвишаващи 1000 върха.



Фигура 1.1.2.1

## Технологии и среди за развой

### Intellij IDEA

Intellij IDEA е интегрирана среда за разработка (на английски: integrated development environment, IDE) за компютърен софтуер създадена от JetBrains. Има две версии - community и ultimate. И двете са подходящи за разработка на софтуер. През 2010 година получава най-висок резултат в тест център спрямо съперниците си Eclipse, NetBeans и JDeveloper. Предлага много функционалности за улеснение на потребителят като рефакториране на код, плъгини и тн.

### Android Studio

Android studio е официалнато IDE за операционната система на Гугъл – Android, изградена върху JetBrains' IntelliJ IDEA софтуера и проектирана специално за разработка на Android приложения. Достъпна е за Windows, macOS и Linux базирани операционни системи. Това е основната среда за разработка на Android.



Фигура 1.2.2.1

### Java

Java е обектно-ориентиран език за програмиране, базиран на класове, специално проектиран, така че да има възможно най-малко зависимости за имплементация. Позволява на програмистите да подкарват програмите си на всякакви платформи. Кода се компилира до т.нар байт код (bytecode), който върви върху виртуалната машина на Java (JVM).



Фигура 1.2.3.1

### Kotlin

Kotlin е по-нов език от Java. Също върви на виртуалната машина на Java и е направен с цел постепенно програмистите да мигрират от Java към него. Използва се основно за разработка на Android приложения. Предлага различни подобрения като "null-safe"оператори, разширени функции и др. Като цяло се стреми да улесни и забърза процеса на разработка.

# Втора Глава

# Функционални изисквания към приложението. Проектиране на редактор на 3D графика

## Функционални изисквания към 3D редактор на графика

* Възможност за вмъкване и съхранение от и към OBJ формат**.**
* Редакция на обектите създадени или вмъкнати в приложението. Това включва:
  + Транслация
  + Скалиране
  + Ротация
  + Промяна на материал
* Да има възможност за undo и redo.
* Йерархична връзка между обектите, която да може да се променя

## Избор на програмен език и библиотеки

### IDE - Android Studio

Предимствата на Android Studio са, че е напълно безплатно и достатъчно за целите на дипломната работа. Приложението ще работи единствено върху Android, така че няма нужда от допълнителни функционалности като тези на Intellij. Средата разполага със всичко нужно като: ProGuard(оптимизира кода), графичен редактор на .xml, система за създаване на .apk инсталационни файлове и тн.

### ViroCore

ViroCore е приложно-програмен интерфейс (API) за Android, алтернативата на SceneKit за iOS. Позволява на Java програмистите да създават 3D сцени за обогатена и виртуална реалност както и обикновени 3D сцени. Докато при други API-та, като OpenGL, човек трябва сам да имплементира методи за рендериране, ViroCore изисква само описание на сцените на високо ниво. Лесно се добавят модели, анимации, ефекти с частици и др. Поддържа работа с най-популярните формати за 3D модели. Сравнително ново API е затова липсват някой функционалности, но за сметка на това често бива актуализирано. Това ще позволи и на текущата дипломна работа да бъде често подобрявана.



Фигура 2.2.2.1

### Език за програмиране - Java

Java е основен език и в Android света. Въпреки че, Android не предлага пълния набор от средства в стандартната библиотека на Java, Android SDK има собствени имплементации на голяма част от тях. Основната причина за този избор е, че ViroCore работи единствено с Java.

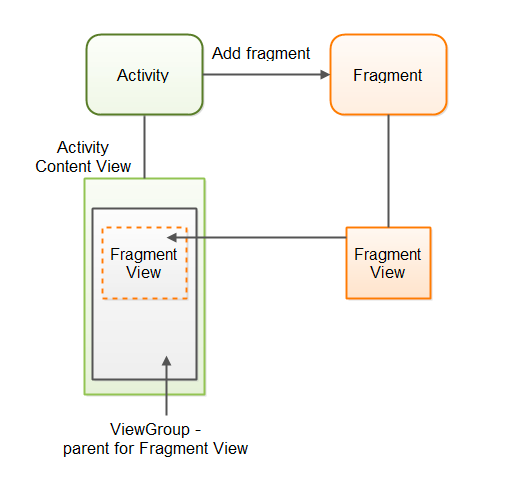
Също така, за разлика от Kotlin, Java е по-разпространен и по-лесно биха се намерили решения на проблеми в сайтове като github и stackoverflow.

## Проектиранe

### Структура на приложението

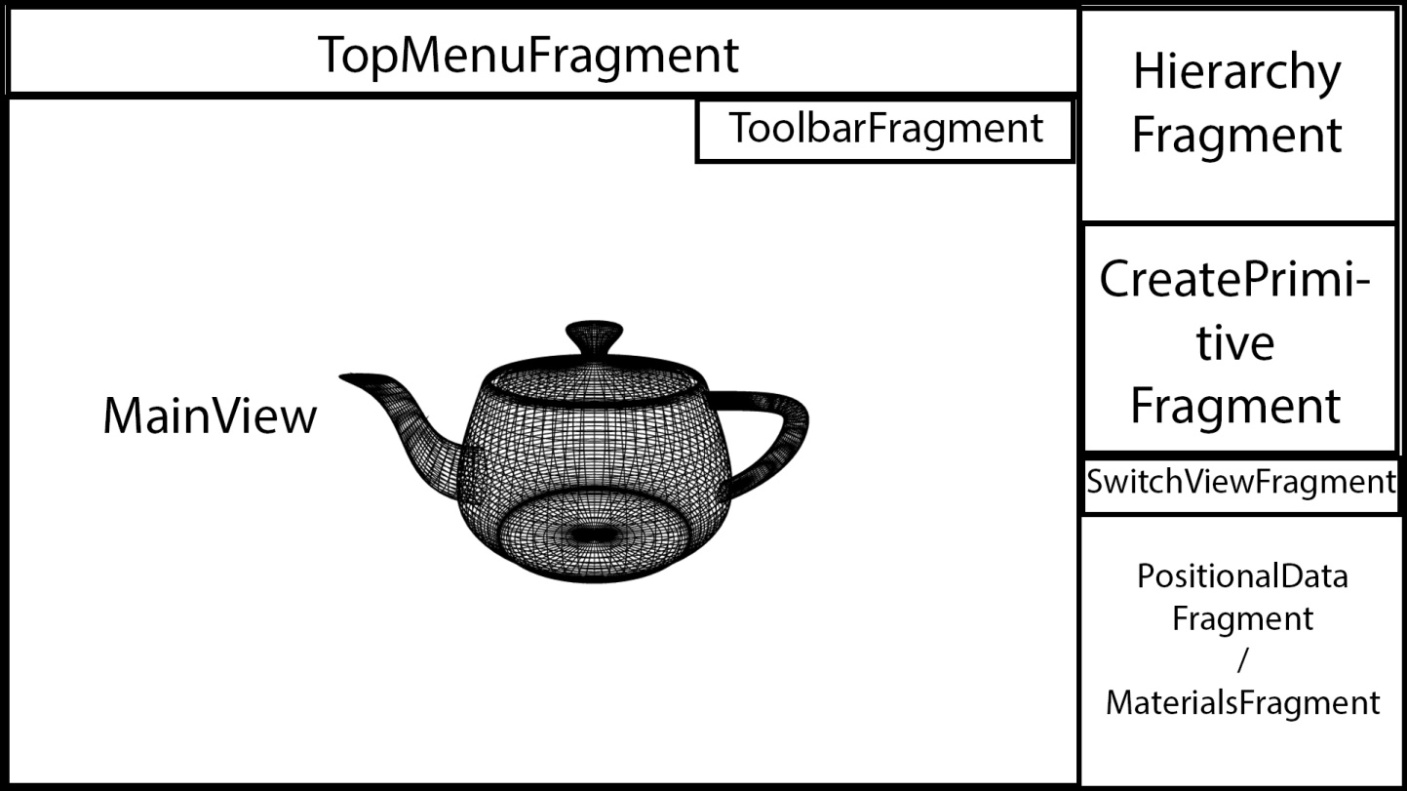
Основната структурна единица на Android приложенията е т.нар. Activity. Най-просто казано то е един екран от вашето приложение, в който потребителят може да взаимодейства с него. Преди е било популярно приложенията да бъдат разделени на множество Activity екрани. В днешно време се препоръчва Android програмите да имат само едно главно Activity. Отделните екрани или елементи на потребителския интерфейс да бъдат групирани във фрагменти.

Фрагментите са най-често част от интерфейса на едно Activity. Те описват собствен изглед в xml файл, който бива вмъкнат в изгледа на Activity-то(*Фигура 2.3.1*.*1*). Ако добавяме даден фрагмент през код, то трябва да има друг изглед, който да се използва за контейнер. Ако не, фрагментът може да бъде описан статично в xml кода на главното Activity. Фрагментите могат да се използват и като отделни екрани. За целта на тази дипломна работа това няма да се налага.



Фигура 2.3.1.1 [[[1]](#footnote-2)]

Приложението ще представлява едно главно Activity поместено във ViroActivity класа. В него ще има един голям изглед на сцената с обекти и множество фрагменти около нея, които представляват отделните части на потребителския интерфейс. *Фигура 2.3.1.2* показва скица на как би изглеждало това заедно с имената на фрагментите.



Фигура 2.3.1.2

### Описание на фрагментите

* CreatePrimitiveFragment - Този фрагмент се състои от няколко бутона, които служат за създаване на примитивни триизмерни обекти като куб, сфера и т.н. Във фрагмента има и бутон за изтриване на селектирания обект.
* TopMenuFragment - Този фрагмент се намира в най-горната част на екрана и предоставя типичните за тези менюта функционалности. Имаме бутони за undo/redo, за вмъкване на модел в сцената и за съхранение.
* HierarchyFragment - В този фрагмент е поместена функционалността за йерархия на обектите. В него се визуализират връзките между обектите. Може лесно да се променя кой обект на кого е дъщерен чрез 'drag and drop'. Също така служи и за по-удобна селекция на обектите ако не можем да ги намерим в сцената.
* PositionalDataFragment - Този фрагмент ни показва основните параметри на всеки обект - позиция, размер и ротация. За всеки от трите параметъра има три стойности съответно за x, y и z осите, които могат да бъдат редактирани и така да бъде променян селектираният обект.
* MaterialsFragment - В MaterialsFragment потребителят може да променя основни компоненти на материала, който използва даден обект. На този етап ViroCore позволява променя на цвят, текстура и модел на осветяване.
* ToolbarFragment - В този фрагмент са поместени допълнителни функционалности и инструменти, които улесняват редакцията на обекти. От там може да се сменя типа на помощните стрелки при селектиране.
* SwitchViewFragment - За да може да се побере повече интерфейс в приложението, някои фрагменти споделят едно и също място (PositionalDataFragment и MaterialsFragment). Превключването между двете става чрез малкия фрагмент над тях - SwitchViewFragment. В него има бутони, чрез които се избира кой фрагмент да е активен.

### Работа с ViroCore

3Д обектите от Viro се визуализират върху специален изглед ViroView. Има различни видове ViroView - за виртуална реалност, за обогатена реалност и за обикновена триизмерна сцена. За целите на дипломната работа ще е нужна сцена от третият тип (ViroViewScene). Всеки такъв изглед може да има сцени, които представляват съвкупността от обекти. Обектите във Viro са пакетирани в класа Node (или Object3D ако е вмъкнат от външен файл). Този клас служи за описване на трансформациите на обекта и връзките му с другите обекти. Всяка сцена си има един главен Node, който служи за център на координатната система. Той се създава автоматично заедно със сцената. Всички останалите обекти трябва да са негови дъщерни.

За самата геометрията на обекта се използва класа Geometry. Всеки Node има по една инстанция от този клас. Тя описва върховете, нормал векторите, материалите. За съжаление текущата версия на ViroCore не позволява достъп до тях (освен материалите). Можем единствено да им задаваме стойности но не и да ги взимаме след това. Това е така защото те се задават в компилирани 'native' java методи, до които потребителят няма достъп. Това ще създаде някои трудности, които са решени по значително заобиколни начини. В официалното хранилище на Viro е споменато, че тази функционалност ще бъде добавена в бъдеще, което ще позволи да се направят по-кадърно някои от нещата по дипломната работа. [[[2]](#footnote-3)]

За да обработва входни събития от потребителя, Viro използва собствени слушатели (Listener), които се задават на ниво Node. Има различни видове Listener интерфейси за различни видове входни данни. Например, за натискане, за влачене, за завъртане и тн.

За да може да има видимост към сцената, тя се нуждае от камера. След като бъде създадена, тя се задава като гледна точка на сцената. Така новият изглед ще показва как изглежда тя от позицията на камерата. Тя също бива поместена в свой Node.

C:\Users\Owner\Desktop\Viro.png

Фигура 2.3.3.1 Структура на ViroCore

# Трета Глава

# Програмна реализация на редактор на 3D графика за Android

## Изграждане на първоначалната сцена

При стартиране на приложението се създава проста сцена, за да не бъде посрещнат потребителят с празен екран. За да се подготви тази сцена, трябва да се направят редица действия.

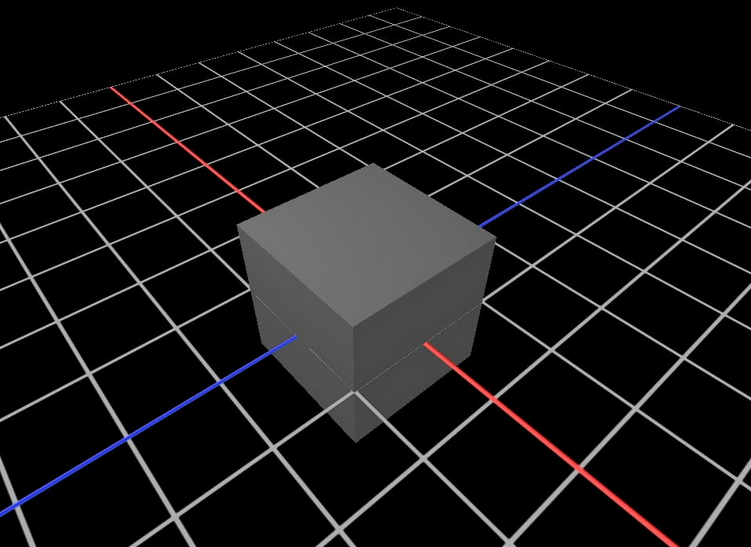
scene = **new** Scene();  
Node rootNode = scene.getRootNode();  
  
Geometry cube = **new** Box(2f,2f,2f);  
..*//init cube*Node cubeNode = **new** Node();  
cubeNode.setGeometry(cube);  
  
Spotlight spotlight = **new** Spotlight();  
AmbientLight ambient = **new** AmbientLight();  
DirectionalLight l = **new** DirectionalLight();  
...*//init lights*Node lightNode = **new** Node();  
...*//add lights*Geometry grid = **new** Quad(16, 16);  
...*//init grid*Node cameraNode = **new** Node();  
camera = **new** OrbitCamera(cameraNode, mainView);  
view.setPointOfView(cameraNode);  
  
rootNode.addChildNode(cubeNode);  
..*//repeat for every node*view.setScene(scene);

Фигура .3.1.1.1

Първо се взима нейният главен Node. Всички останали обекти се вмъкват като негови дъщерни. Поставя се куб с размери 2х2х2 в центъра на света. Създава се Node, който да отговаря за светлината. В него има три светлини - една от околната среда, една прожекторна и една насочена светлина.

За да има някакъв ориентир е добре да има решетка легнала върху хz равнината. Използва се текстура на решетка 16х16, чиито централни перпендикулярни линии са оцветени в червено за оста х и синьо за z. Тази текстура се слага върху равнина, създадена от Viro, и се позиционира на нужното място. Потребителят не може по никакъв начин да взаимодейства с нея. Тя служи единствено като ориентир.

Накрая се създава камера, която да използваме за гледна точка. Когато всичко е готово сцената изглежда така:



Фигура 2.3.3.1.3.1.2

## Обработване на входни събития от фрагментите.

За да може един изглед да реагира на входни събития, то на него трябва да му бъде зададен слушател, който имплементира Listener интерфейс. Например, OnClickListener ако е бутон за натискане, или TextWatcher ако е текстово поле и т.н. За да се спести писане на код, както и за да е по-разбираем кодът, е добре да се използва един Listener клас вместо да се дефинират различни за всеки един бутон или поле. Популярна практика е самият фрагмент да имплементира нужните Listener интерфейси. При създаването на фрагмент (оnCreateView();) на съответните бутони им бива зададен като Listener да бъде фрагментът (*Фигура 3.2.1*).

**public** View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {  
 **final** View view = inflater.inflate(R.layout.***fragment\_create\_primitive***, container, **false**);  
 Button createCube = view.findViewById(R.id.***createCube***);  
 createCube.setOnClickListener(**this**);=  
 Button deletebutton = view.findViewById(R.id.delete);  
 deletebutton.setOnClickListener(**this**);  
...  
 **return** view;  
}

Фигура 3.2.1

След това във функциите за обработка на събитието се използва switch оператор, чрез който се разбира кой бутон е бил натиснат като проверяваме според неговото id, което сме задали в xml кода на фрагмента.  
 Повечето фрагменти в текущата дипломна работа използват именно този метод за обработка на входни събития затова е добре да бъде обяснен още в началото.

## Undo/Redo

Друга функционалност разпространена из цялото приложение е тази за връщане назад и напред по действията на потребителят (undo/redo). Един истински редактор трябва да има такава възможност. За да се постигне това, всяко действие на потребителя, което се счита за редакция, трябва да бъде записано. На *фигура 3.3.1* е показан абстрактният клас, който се използва за записване на действията на потребителя. Той има единствено две функции, описващи какво прави действието в случай на undo и на redo. Не е нужно нищо друго защото действията може да са най-различни и всичко останало зависи от ситуацията.

**public abstract class** Action {  
 **public abstract void** executeUndo();  
 **public abstract void** executeRedo();  
}

Фигура 3.3.1

Имплементирани са различни класове за различните действия, които може да изпълни потребителя. Например, класът за действие на преместване на обект в пространството (*Фигура 3.3.2*) ще приема един Node, който е бил преместен, и два вектора, съответно старата и новата позиция на обекта. В метода за undo, позицията на обекта се задава да е равна на старата му позиция, а в този за redo на новата.

**public** TranslateAction(Node node, Vector prevPosition, Vector newPosition) {  
 **this**.node = node;  
 **this**.prevPosition = prevPosition;  
 **this**.newPosition = newPosition;  
}  
  
@Override  
**public void** executeUndo() {  
 node.setPosition(prevPosition);  
}  
  
@Override  
**public void** executeRedo() {  
 node.setPosition(newPosition);  
}

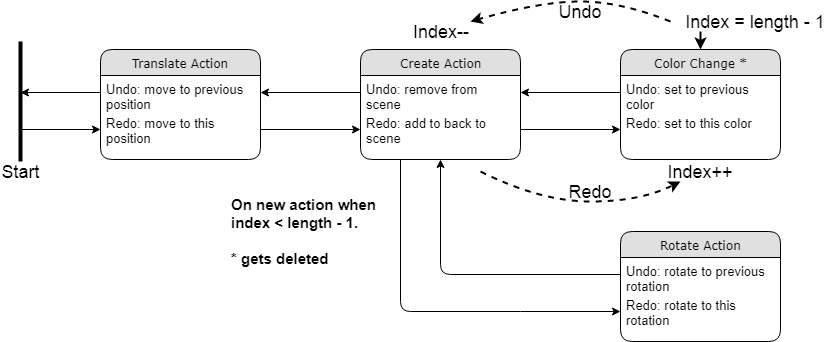
Фигура 3.3.2

Запазването на действията в даден ред и обхождането им чрез undo/redo става с помощта на класа ActionsController (*Фигура 3.3.3*) .

**public void** addAction(Action action) {  
 **if**(index == actions.size()) {  
 actions.add(action);  
 index++;  
 } **else** {  
 actions.subList(index, actions.size()).clear();  
 addAction(action);  
 }  
}  
  
**public void** undo() {  
 **if**(index > 0) {  
 index--;  
 actions.get(index).executeUndo();  
 }  
}  
  
**public void** redo() {  
 **if**(index <= actions.size() - 1) {  
 actions.get(index).executeRedo();  
 index++;  
 }  
}

Фигура 3.3.3

Променливата index показва къде се намира потребителят в момента в списъка от действия. Той се намаля при извикване на undo и се увеличава при redo. Има и граници, в които тези функции могат да бъдат извиквани. Няма как да се изпълни undo ако индексът е нула или да се изпълни redo ако индексът сочи към последния елемент от списъка. Така няма как да се излезе от неговите граници. Също така при добавяне на ново действие, в случай че потребителят не се намира в края на списъка, трябва да бъдат заличенивсички действия след текущата позиция и да се започне добавяне от нея.



Фигура . Схема на ActionsController

## Създаване и изтриване на примитивни обекти

Фрагментът CreativePrimitiveFragment съдържа в себе функционалност за създаване и изтриване на обекти. Обектите, които могат да бъдат създавани са готови примитивни обекти, моделирани на Blender, които се вмъкват от 'asset' папката на проекта. Не се използват готовите обекти от ViroCore, защото те не дават достъп до данните на модела си. Това ще направи тяхното съхранение невъзможно. Класът OBJObject запазва тези данни като ги извлича от текстовото представяне на обекта. Той разширява класа Object3D, който се използва за зараждане на външен обект. В зависимост от това кой бутон е бил натиснат, програмата създава различен обект използвайки предварително готовите модели*.*

**case** R.id.createPyramid:  
 String text = readFromAsset(**"def\_pyramid.obj"**);  
 Uri uri = Uri.parse(**"file:///android\_asset/def\_pyramid.obj"**);  
 OBJObject obj = **new** OBJObject(ViroActivity.getView().getViroContext()  
 , uri, text, **this**);

Фигура 3.4.1

В конструктора на OBJObject се изваждат всички нужни данни от текста на файла. Форматът OBJ е прост. В зависимост от това какъв е първият символ от реда се разбира дали отговаря за връх, нормал, лице и т.н. Тези данни биват записани в спицъци, с изключение на лицата, тъй като те няма да бъдат редактирани. Тях запазваме като обикновен текст, който да използваме при съхранение.

**switch** (element) {  
 **case "v "**:  
 addEntryTo(vertices, line); **break**;  
 **case "vt"**:  
 addEntryTo(textureCoords, line); **break**;  
 **case "vn"**:  
 addEntryTo(normals, line); **break**;  
 **case "f "**:  
 faces.append(line).append(ln); *//ln for new linebreak;* **case "o "**:  
 setName(line.substring(2)); **break**;  
 }  
 ...  
**private void** addEntryTo(List<Vector> list, String line) {  
 String[] entry = line.split(**"\\s+"**);  
 list.add(**new** Vector(  
 Float.parseFloat(entry[1]),  
 Float.parseFloat(entry[2]),  
 Float.parseFloat(entry[3])  
 ));  
}

Фигура 3.4.2

Това обаче дава само един статичен модел. За да може той да бъде правилно вмъкнат в сцената, така че потребителят да може да взаимодейства с него, се използва addToScene().

Тя изпълнява всички други нужни действия за коректното вмъкване на нов обект в програмата. Първо му поставя материал и име. Взима се ViroActivity, за да може новосъздадения Node да се подаде на неговата makeNodeSelectable() функция. Тя задава на Node обекта слушател, който при натискане да го селектира. Функцията sleectNode() е обяснена подробно в точка 3.6. Така той ще може да бъде правилно селектиран от потребителя. След това се прави дъщерен на главния Node в сцената.

**public void** makeNodeSelectable(Node node) {  
 node.setClickListener(**new** ClickListener() {  
 @Override  
 **public void** onClick(**int** i, Node node, Vector vector) {  
 selectNode(node);  
 }

Фигура 3.4.3

В случай че искаме да запишем създаването като действие, което да може да се върне назад го записваме като такова. В някои случаи това не трябва да се прави. Например, ако го използваме за вмъкване на някой служебен обект или го извикваме от самия ActionsController с цел да върнем изтриване. Ако бъде записан отново ще има повторение.

Накрая се взима фрагментът отговарящ за йерархията и обектът добавя и към нея на нулево ниво, т.е не е дъщерен обект на друг(йерархичната структура на обектите е обяснена подробно в 3.8).

**public void** addToScene(Node n, String name, **boolean** recordAction) {  
 n.getGeometry().setMaterials((  
 Arrays.*asList*(MaterialsFragment.*getMaterials*().get(0))));  
 n.setName(name);  
 ViroActivity activity = (ViroActivity) getActivity();  
 activity.makeNodeSelectable(n);  
 ViroActivity.*getScene*().getRootNode().addChildNode(n);  
  
 **if**(recordAction) {  
 ActionsController.*getInstance*().addAction(**new** CreateAction(n));  
 }  
  
 FragmentManager fm = activity.getFragmentManager();  
 HierarchyFragment hierarchy = (HierarchyFragment)

fm.findFragmentById(R.id.***hierarchyFragment***);  
 hierarchy.addToHierarchy(n, 0);  
}

Фигура 3.4.4

За изтриването ViroCore разполага с метод disposeAll(), който изтрива всички ресурси свързани с обекта (материали, геометрия, тн.). Той, обаче, не може да бъде използван поради две причини. Първо, тъй като обектите споделят материали, изтриването на обект по този начин би изтрило и материала на останалите обекти използващи същия като него. Второ, така не би имало начин да се върне това действие чрез undo, защото всички данни ще бъдат загубени. Затова изтриването става по друг начин. Това, която се прави е обекта да бъде премахнат от родителя си. Без родител обектът реално не принадлежи към сцената, затова изчезва. Неговите ресурси не се изтриват и могат да бъдат запазени, за да бъде добавен отново при нужда.

**case** R.id.Delete:  
 Node selected = ViroActivity.getSelectedNode();  
 ActionsController.getInstance().addAction(

**new** DeleteAction(selected, selected.getParentNode()));  
 selected.removeFromParentNode();  
 **return**;

Фигура 3.4.5

## Основни параметри на обекта

Всеки обект намиращ се в сцената има три основни параметъра, които определят неговото разположение в триизмерното пространство. Това са позиция, размер и ротация. Фрагментът PositionalDataFragment показва какви са те за даден обект и дава възможност за тяхното редактиране. Има по три стойности за всеки един от параметрите. Те отговарят съответно за x, y и z компонентите. За да може обекта реално да се промени при редакция на текстовото поле, трябва да имплементираме поведението на всяко от полетата при промяна. За да се опише това поведение трябва да се използва интерфейсът TextWatcher. Класовете PosTextWatcher, ScaleTextWatcher, RotationTextWatcher имплементират този интерфейс. *Фигура 3.5.1* показва как се задават техните инстанции, така че да следят текстовите полета за позиция. Масивът от цели числа служи за да се разбере по коя ос трябва да бъде нанесена промяната след това. Ако има единица на първи елемент значи по x оста и т.н.

**this**.xPos = view.findViewById(R.id.xPosition);  
**this**.yPos = view.findViewById(R.id.yPosition);  
**this**.zPos = view.findViewById(R.id.zPosition);  
xPos.addTextChangedListener(**new** PosTextWatcher(**new int**[]{1,0,0}));  
yPos.addTextChangedListener(**new** PosTextWatcher(**new int**[]{0,1,0}));  
zPos.addTextChangedListener(**new** PosTextWatcher(**new int**[]{0,0,1}));

Фигура 3.5.1

Самата промяна в обекта става в метода afterTextChanged(Editable s), който се извиква след промяна в текста на дадено поле, показан на Фигура 3.5.2.

**public void** afterTextChanged(Editable s) {  
 Node selected = ViroActivity.*getSelectedNode*();  
 Vector currentPosition = selected.getPositionRealtime();  
 Vector newPosition = getNewValue(currentPosition,  
 Float.*parseFloat*(s.toString()), axis);  
 selected.setPosition(selected.getParentNode()  
 .convertWorldPositionToLocalSpace(newPosition));  
  
 ActionsController.*getInstance*().addAction(**new** TranslateAction(  
 selected, currentPosition, selected.getPositionRealtime()));  
}  
...  
**protected** Vector getNewValue(Vector old, **float** value, **int**[] axis) {  
 Vector diff = **new** Vector((value - old.x) \* axis[0], (value - old.y) \* axis[1],(value - old.z) \* axis[2]);  
  
 **return** old.add(diff);  
}

Фигура 3.5.2

За да може кода да е достатъчно абстрактен и да работи и за трите оси, както и за трите параметъра(позиция, размер и ротация), се използва функцията getNewValue(), която ще пресметне разликата между сегашните стойности на обекта и новите. Умножават се по целочисления масив, служещ за разпознаване на оста, за да се получи стойност само по тази ос, която трябва да се промени. След това е нужно просто да бъде добавена тази разлика към досегашните стойности. При задаване на новата позицията е важно тя да бъде трансформирана към локални координати, тъй като обектът може да е дъщерен на друг. Тогава неговата позиция се смята спрямо неговия родител.

Накрая се добавя действие в ActionsController за да може потребителят да върне назад тази промяна. Кодът е аналогичен при промяна на размер и ротация.

## Селектиране на обект

Селектирането на обект е важна част от приложението, тъй като всички действия, които прави потребителят трябва да се изпълняват именно върху текущия обект. Той се запазва в статична променлива, която лесно може да бъде достъпвана от всяка част на приложението. За да бъде възможно селектирането на даден Node, той трябва да има нужния Listener, който при натискане ще извика функцията selectNode(). Първото, което тя е прави е да провери дали текущия Node не е вече селектиран. В такъв случай няма да има нужда да се изпълни. След това записва натиснатият Node в променливата selectedNode и прилага помощни стрелки към него. Накрая обновява изгледа с параметри на обекти.

**public void** selectNode(Node node) {  
 **if**(selectedNode != node) {  
 **if**(activeHandles == **null**) {  
 activeHandles = **new** TranslateHandles(node);  
 } **else** {  
 activeHandles.setParent(node);  
 }  
 selectedNode = node;  
  
 ObjectParamsFragment paramFrag = (ObjectParamsFragment)  
 getFragmentManager().findFragmentById(  
 SwitchViewFragment.*getCurrentId*());  
 paramFrag.update(selectedNode);  
 }  
}

Фигура 3.6.1

## Помощни стрелки при селектиране

Всеки един редактор на 3D графика разполага с тази функционалност, за да може потребителят да променя позицията, размера и ротацията на даден обект директно от сцената. Тези стрелки са перпендикулярни една на друга и успоредни на осите на координатната система. Изключение са помощните стрелки за ротация, които на практика не са стрелки а три окръжности около обекта.

Тъй като трите вида стрелки изпълняват подобни функции техните класове са дъщерни на абстрактния клас Handles *Фигура 3.7.1*. Това, което той прави е да позиционира стрелките в пространството, да им зададе различни материали за различните оси и да осъществи връзката със селектирания обект(техен родител/parent). Важно е да се отбележи, че помощните стрелки не са свързани като деца на обекта през Node структурата на ViroCore. От една страна това би улеснило процеса като автоматично ги премества заедно със селектирания обект, но от друга би попречило по много начини. Стелките ще се завъртат и скалират заедно с него, което ще ги направи несиметрични и не успоредни на x, y, z осите. Поради тази причини това не се прави и позицията на стрелките се наглася ръчно, така че да следва обекта родител.

Моделите, които използват помощните стелки са изработени на Blender и запазени в OBJ формат в 'asset' папката на приложението. Така лесно могат да бъдат заредени чрез loadModel() функцията от ViroCore.

Handles(ViroView view, String handleAssetPath, Node parent) {  
...*//set names, etc* **this**.parent = parent;  
 rootNode.addChildNode(handleRoot);  
 handleRoot.setPosition(  
 parent.getWorldTransformRealTime().extractTranslation());  
...*//initHandles*}  
  
**private void** initHandle(**final** Object3D handle, ViroView view, String handleAssetPath, **final** Vector rotation, **final int** color) {  
 handleRoot.addChildNode(handle);  
 handle.setRotation(rotation);  
 handle.loadModel(view.getViroContext(), Uri.*parse*(handleAssetPath), Object3D.Type.OBJ, **new** AsyncObject3DListener() {  
 **void** onObject3DLoaded(Object3D object, Object3D.Type type){  
 ...*//add proper material* }  
 });  
}

Фигура 3.7.1

Съществената част на всяка от помощните стрелки е в нейния OnDragListener и OnClickListener. Случаят на транслиране е най-прост. След като се види по коя ос трябва да бъде преместен обекта, просто му се създава нова позиция променена само по тази ос. Тя трябва да бъде конвертирана в локални координати, тъй като обекта може да е дъщерен на някой друг. Нещо важно е това, че в момента ViroCore не предлага вариант за влачене на обект само по една ос, но има такъв за равнина. Стелките са настроени по този начин и това означава, че те могат да се изместят настрани от обекта. Затова постоянно се нулира позицията (0,0,0). Това понякога не работи перфектно и стрелката прескача от обекта до позицията на пръста и обратно много бързо. При въвеждане на тази функционалност този бъг може лесно да бъде оправен.

Нещата, които трябва се правят само веднъж се намират в OnClickState функцията. При първоначално докосване CLICK\_DOWN, трябва да бъде запазена досегашната позиция. При отпускане на пръста CLICK\_UP отново връщаме стрелката в начална позиция. Накрая запазваме действие за транслиране със старата позиция, която беше запазена в началото и сегашната.

@Override  
**public void** onDrag(**int** i, Node node, Vector local, Vector world) {  
...*//get newPos by taking only one axis of world* parent.setPosition(parent.getParentNode()  
 .convertWorldPositionToLocalSpace(newPos));  
 handleRoot.setPosition(newPos);  
 node.setPosition(**new** Vector(0f, 0f, 0f));  
}  
  
@Override  
**public void** onClickState(**int** i, Node node, ClickState clickState, Vector vector) {  
 **if**(clickState.equals(ClickState.CLICK\_DOWN)) {  
 oldPos = parent.getWorldTransformRealTime().extractTranslation();  
 }**else if**(clickState.equals(ClickState.CLICK\_UP)) {  
 Vector newPos = parent.getWorldTransformRealTime().extractTranslation();  
 getHandleRoot().setPosition(newPos);  
  
 node.setPosition(**new** Vector(0f, 0f, 0f));  
 ActionsController.*getInstance*().addAction(  
 **new** TranslateAction(parent, oldPos, newPos));  
 }  
}

Фигура 3.7.2

Стрелките за ротация и скалиране са имплементирани по подобен начин.

## Йерархия

C:\Users\Owner\Desktop\hierachy.png

Фигура 3.8.1

За по-лесно проследяване на обектите в сцената, софтуерите за 3D обработка имат изглед с йерархията на обектите в сцената. Така лесно може да се манипулират връзките между тях, например кой обект на кого е дъщерен, както и да бъдат селектирани директно от там ако не може да се намерят в сцената. *Фигура 3.8.1* показва примерна йерархия. От нея се разбира, че Cube е дъщерен обект на Sphere, а Base и Top на Pyramid.

Функционалността на йерархията е имплементирана във фрагмента HierarchyFragment. За изписване на всеки един от обектите се използва TextView класът. Всяко TextView е свързано със съответния Node от сцената в един HashMap. Тази връзка позволява лесното селектиране на правилния Node в сцената, при натискане на даденo TextView.

Йерархията се попълва с помощта на функцията addToHierarchy(Node node, int level) *Фигура 3.8.2*. Създава се ново TextView. Текстът му се задава да бъде името на обекта, който бива добавен. Преди името трябва да се вмъкнат толкова на брой символа "->", колкото е променливата level. Така ще може да се онагледи кога един обект е дъщерен на друг.

**public void** addToHierarchy(Node node, **int** level){  
 TextView name = **new** TextView(ViroActivity.*getView*().getContext());  
  
 name.setText(contructName(node, level));  
 hierarchy.addView(name);  
  
 name.setOnClickListener(**this**);  
 name.setOnLongClickListener(dragAndDropListener);  
 name.setOnDragListener(dragAndDropListener);  
  
 nodes.put(name, node);  
}  
  
**private** String contructName(Node node, **int** level){  
 String tabs = **new** String(**new char**[level]).replace(**"\0"**, **"->"**);  
  
 **return** String.*format*(**"%S%s%s"**, **"| "**, tabs, node.getName());  
}

Фигура 3.8.2

Задавайки слушателя при натискане на TextView обекта да е самия фрагмент, отново се използва познатия метод за обработка на събития. В onClick() функцията (*Фигура 3.8.3*) на текстовото поле първо се намира правилния Node, използвайки натиснатото TextView като ключ във свързващия HashMap. Селектира се чрез selectNode() функцията от ViroActivity.

@Override  
**public void** onClick(View view) {  
 TextView element = (TextView) view;  
 Node selection = nodes.get(element);  
  
 ViroActivity activity = (ViroActivity) getActivity();  
 activity.selectNode(selection);  
}

Фигура 3.8.3

Освен селектиране на обектите йерархията дава функционалност и за промяна на връзките между обектите. Това става чрез обикновен 'drag and drop'. Изгледа, който бъде довлачен до друг изглед му става дъщерен. За да бъде постигната тази функционалност е дефиниран вътрешен клас DragAndDropListener имплементиращ интерфейсите OnLongClickListener и OnDragListerner.

Събитието за влачене започва при дълго натискане върху обекта, за да може да се различава от нормално кратко натискане за селекция Фугира 3.8.4. Shadow представлява как би изглеждала "сянката" на изгледа при влаченето. В случая на конструктора е подаден самия изглед затова тя би изглеждала като самото TextView на обекта.

@Override  
**public boolean** onLongClick(View view) {  
 ClipData data = ClipData.newPlainText(**""**, **""**);  
 View.DragShadowBuilder shadow = **new** View.DragShadowBuilder(view);  
 view.startDrag(data, shadow, view,0);  
 **return true**;  
}

Фигура 3.8.4

onDrag() Фигура 3.8.5 функцията се интересува само от крайния момент, когато потребителят премахва пръста от екрана си. Това става когато събитието достигне ACTION\_DROP, затова в нея се проверява единствено за него. Когато това стане трябва да се свършат доста неща. Първо трябва да се определи новата позиция, на която да бъде преместен текстовият изглед на дъщерния обект. Първият цикъл итерира през всички обекти във йерархията докато не намери този, върху когото е бил довлачен пръстът на потребителя. Следователно нужната позиция е тази на обекта(бащиния) плюс едно. След това изгледа се премахва от старата му позиция и се добавя на новата.

Важно е да се добавят толкова "->" символа, колкото са тези на бащиния плюс едно. Така потребителят ще разбере, че наистина довлаченият обект е станал дъщерен на другия. Разликата между дължината на текста на бащиния обект със "->" и без дава колко са неговите стрелки. Тези на дъщерния са толкова плюс едно. Тази разлика се дели на две, тъй като съчетанието "->" реално е съставено от два символа и за всяко негово съвпадение ще има разлика от две.

След това дъщерният обект се откача от стария му родител и се закача за новия. Трябва да се направи и промяна в позицията на дъщерния обект, тъй като тя се възприема като локална спрямо обекта родител. Така например ако имаме обект, който се намира в центъра на координатната система (вектор 0, 0, 0) и го направим дъщерен на друг обект, тогава той би се преместил върху неговата позиция, защото досегашните локални координати вече отговарят на позицията на обекта родител. Можем да разберем новата нужна позиция като обърнем досегашните координати на дъщерния обект спрямо света в локални за обекта родител с функцията convertWorldPositionToLocalSpace().

Накрая окончателно се задава текстът на изгледа. Функцията constructName създава този текст използвайки името на обекта и толкова на брой "->" колкото са нужни.

**case** DragEvent.ACTION\_DROP:  
 TextView container = (TextView) v;  
 **int** newIndex = 0;  
 **int** childCount = hierarchy.getChildCount();  
 **for**(**int** i = 0;i < childCount;i++) {  
 **if**(container.equals(hierarchy.getChildAt(i))) {  
 newIndex = i + 1;  
 }  
 }  
 hierarchy.removeView(target);  
 hierarchy.addView(target,Math.min(newIndex,childCount -1));  
   
 String text = container.getText().toString();  
 String temp = text.replace(**"->"**, **""**);  
 **int** level = ((text.length() - temp.length()) / 2) + 1;  
   
 Node parent = nodes.get(container);  
 Node child = nodes.get(target);  
   
 Vector newPos = parent.convertWorldPositionToLocalSpace(  
 child.getPositionRealtime());  
 child.removeFromParentNode();  
 parent.addChildNode(child);  
 child.setPosition(newPos);  
   
 target.setText(contructName(child, level));  
 **break**;  
 ...  
**private** String contructName(Node node, **int** level){  
 String tabs = **new** String(**new char**[level]).replace(**"\0"**, **"->"**);  
 **return** String.format(**"%S%s%s"**, **"| "**, tabs, node.getName());  
}

Фигура 3.8.5

## Промяна на материала на обекта

В момента на писане на тази дипломна най-новата версия на ViroCore е 1.13. Възможните параметри на материал, които могат да бъдат променяни, са цвят, текстура, модел на осветяване и някой други по-незначителни. Във фрагмента MaterialsFragment има функционалност за редактиране на споменатите три. Методът на редактиране и запазване на материалите е подобен на този в Blender. Потребителят може да създава множество материали, които се запазват, за да могат лесно да бъдат преизползвани върху други обекти. Разликата е, че при Blender новосъздадените обекти нямат зададен материал, докато в настоящата дипломна работа те започват с един и същи материал по подразбиране. Ако потребителят го промени, новите обекти, които създава ще се създадат с редактирания материал.

За да може всичко това да се проследява и запазва са нужни определени променливи в класа на MaterialsFragment показани на *Фигура 3.9.1.*

**private** View **colorView**;  
**private** TextView **hexColor**;  
**private** Material.LightingModel[] **lightingModels** =  
 **new** Material.LightingModel[]{*//literal of all light models...};***private** Spinner **lightModelSpinner**, **materialsSpinner**;  
**private static** ArrayList<Material> *savedMaterials* = **new** ArrayList<>(Arrays.*asList*(*makeDefaultMat*()));  
**private** Material **selectedMat**;  
**private** ArrayAdapter<String> **materialAdapter**;

Фигура 3.9.1

Изгледът colorView представлява малък цветен квадрат, който показва текущия цвят, а hexColor е същия цвят представен като текст в шестнадесетичен вид. Масивът lightnightModels съдържа в себе си всички варианти на модел на осветяване поддържани от ViroCore. Чрез него може правилно да бъде зададен модела в зависимост от избрания елемент от lightModelSpinner. Класът Spinner представлява обикновено падащо меню. Имаме едно и за избор на текущия материал - materialsSpinner. Новосъздадените материали се запазват в списъка savedMaterials. А за да може промените, които въвежда потребителят да се извършват върху правилния материал, селектирания в момента материал се запазва в selectedMat. Накрая имаме ArrayAdapter<String>.

По принцип класът ArrayAdapter се използва за създаване на изглед на всеки елемент от масива. В случая той се използва за попълване на materialsSpinner. Spinner обектите имат нужда от такъв адаптер за да показват правилно елементите си. При създаване на нов материал, той бива добавен към materialsSpinner, чрез адаптера. Адаптера на падащото меню за модел на светлина няма нужда да бива запазен, тъй като то не се променя. Използваме го само веднъж при създаване на менюто.

За избор на цвят се използва средство взето от сайта <https://android-arsenal.com> [[[3]](#footnote-4)]. Когато цвета бъде избран (Фигура 3.9.2) той бива зададен върху материала и съответното действие бива записано в ActionsController. Накрая изгледите във фрагмента трябва да бъдат обновени за да отразят промяната в цвета.

@Override  
**public void** onColorPicked(**int** color) {  
 ActionsController.*getInstance*().addAction( **new** ChangeColorAction(  
 **selectedMat**.getDiffuseColor(), color, **selectedMat**));  
  
 **selectedMat**.setDiffuseColor(color);  
 update(**selectedMat**);  
}

Фигура 3.9.2

Функцията update(Material mat); Фигура 3.9.3 прави всички нужни промени върху изгледите. Променя цвета на ColorView изгледа, който го показва, сменя текста на шестнадесетичния вид на цвета и променя избрания елемент на двете падащи менюта. Също така, в началото на функцията селектирания материал се задава да бъде равен на материала, който бива подаден като аргумент. Това се прави в случай, че обновяването става с чисто нов материал, например при самото създаване. Тогава следващите промени трябва да бъдат нанасяни върху него. Тази функция се извиква на всички места, където има промяна върху материала.

**public void** update(Material mat) {  
 **selectedMat** = mat;  
 **int** color = mat.getDiffuseColor();  
 **colorView**.setBackgroundColor(color);  
 **hexColor**.setText(colorHex(color));  
  
 Material.LightingModel lightModel = mat.getLightingModel();  
 **for**(**int** i = 0;i < **lightingModels**.**length**;i++) {  
 **if**(lightModel.equals(**lightingModels**[i])) {  
 **lightModelSpinner**.setSelection(i);  
 }  
 }  
 String name = mat.getName();  
 **for**(**int** i = 0;i < **materialAdapter**.getCount();i++) {  
 **if**(name.equals(**materialAdapter**.getItem(i))) {  
 **materialsSpinner**.setSelection(i);  
 }  
 }  
}

Фигура 3.9.3

За да се сложи текстура на материала се използва Intent. Най-просто казано това е намерението на програмата да извърши някакво действие. Типът Intent.ACTION\_GET\_CONTENT ще отвори нов екран, от където потребителят ще може да избере файл. След това във функцията onActivityResult (Фигура 3.9.4) резултатът от този избор може да бъде извлечен във Uri обект. При получаване на резултата се създава InputStream, който да чете от получените данни. След това, както при слагането на координатната мрежа, той се превръща в Bitmap. Така вече може да бъде използван като текстура.

Intent **intent** = **new** Intent(Intent.***ACTION\_GET\_CONTENT***);  
 intent.setType(**"image/\*"**);  
 intent.addCategory(Intent.CATEGORY\_OPENABLE);  
 startActivityForResult(Intent.createChooser(  
 intent, **"Select a File to Upload"**), 1);  
 ...  
**public void** onActivityResult(**int** requestCode,**int** resCode,Intent data){  
 **super**.onActivityResult(requestCode, resultCode, data);  
 Uri uri = data.getData();  
  
 InputStream i = getActivity()  
 .getContentResolver().openInputStream(uri);  
 Bitmap bitmap = BitmapFactory.decodeStream(i);  
 Texture t = **new** Texture(bitmap, Texture.Format.RGBA8, **true**, **false**);  
 ActionsController.getInstance().addAction( **new** ChangeTextureAction(  
 selectedMat.getDiffuseTexture(), t, selectedMat));  
 selectedMat.setDiffuseTexture(t);  
}

Фигура 3.9.4

## Орбитираща камера

За да се движи из сцената потребителят ще се нуждае от камера, която е фокусирана и орбитира около дадена точка. ViroCore разполага със специална камера с подобни настройки. Тя, обаче, работи само при виртуална и обогатена реалност. Във въпрос №30 на официалното им хранилище е предоставена имплементация на орбитираща камера за обикновени 3D сцени. [[[4]](#footnote-5)]. Нейното поведения задоволява само част от нужните изисквания. Класът OrbitCamera представлява разширение на описаната камера. Освен завъртане тя има приближаване/отдалечаване и преместване настрани(паралакс).

За имплементацията на приближаване се използва вградения детектор на мащабиращо движение (два пръста се отдалечават или приближават). В началото на движението се запазват досегашните ъгли описващи позицията на камерата спрямо фокусната точка. След това функцията onScale директно дава коефициент представляващ стойността на скалирането. Радиусът на камерата се умножава по този коефициент и новата позиция на камерата се пресмята със същите началните ъгли. В случай, че имаме селектиран обект умножаваме и размера на помощните стрелки по този коефициент, за да може да са видими дори при много отдалечен обект. Коефициента бива изваден от 2, защото иначе отдалачеването би ставало при свиване на пръстите, приближаването при отдалачаване, а трябва да е наобратно.

**public boolean** onScale(ScaleGestureDetector detector) {  
 **float** scaleFactor = 2 - detector.getScaleFactor();  
 radiusConst \*= scaleFactor;  
 Vector newPos = getPositionFromAngles(theta, phi);  
 setCameraPosition(newPos, lookAt);  
 **if**(hasSelected){  
 activeHandles.setScale(activeHandles.getScaleRealtime()  
 .scale(scaleFactor));  
 }  
 **return true**;  
}

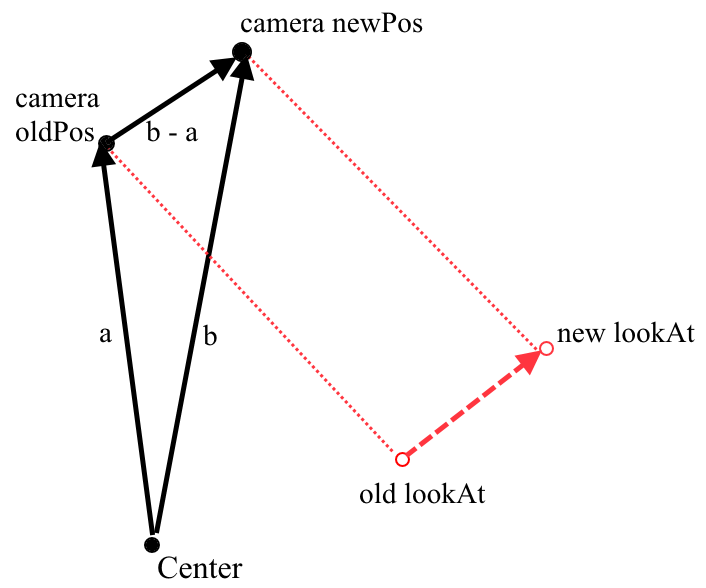
Фигура 3.10.1

За да се постигне ефектът на паралакс е нужно позицията на камерата и тази на фокусната точка да бъдат изместени настрани по локалните 'ху' координати на камерата. Това се случва когато потребителят направи плъзгащо движение с два пръста. В Android няма вградено подобно движение, но съществуват множество библиотеки предоставящи допълнителни движения. За тази цел е използвана библиотека от Almeros [[[5]](#footnote-6)]. Използвайки OnMoveGestureDetector тази разлика в x и y координатите може лесно да бъде достъпена.

**float x** = -d.x \* 0.01f;  
**float y** = d.y \* 0.01f;  
Vector **oldPos** = cameraNode.getPositionRealtime();  
Vector **newPos** = **new** Vector(**x**, **y**, 0f);  
Vector cameraPos = **cameraNode**.convertLocalPositionToWorldSpace(newPos);  
**lookAt** = **lookAt**.add(cameraPos.subtract(oldPos));  
setCameraPosition(cameraPos, **lookAt**);

*Фигура 3.10.2*

След като тя бъде извлечена, се създава нов вектор с тези разлики като координати и нула по 'z'. Те могат да се умножат по коефициент за да се намали или увеличи размера на ефекта. Предишната позиция се запазва, а новата се пресмята спрямо локалните координати на камерата. Тъй като фокусната точка не е Node, а просто вектор, тя трябва да бъде изместена по друг начин. Изваждайки новата и старата позиция на камерата се получава вектора, по който трябва да се транслира точката, така че да се придвижи паралелно на камерата.



Фигура 3.10.3

Самото преместване на камерата става в setCameraPosition() функцията. Тя пресмята позицията използвайки радиуса, азимутен ъгъл и ъгъл за височина. Това, което е променено в сравнение с [4], е че накрая се добавя вектора, представляващ фокусната точка. Ако това не се направи камерата само ще е обърната към нея, но ще орбитира центъра на координатната система.

*// Parametrize the camera's location onto a sphere based on current phi and theta values.***private** Vector getPositionFromAngles(**double** theta, **double** phi) {  
 **double** camZ = **radiusConst** \* Math.*cos*(Math.*toRadians*(theta)) \*

Math.*sin*(Math.*toRadians*(phi));  
 **double** camX = **radiusConst** \* Math.*sin*(Math.*toRadians*(theta)) \*

Math.*sin*(Math.*toRadians*(phi));  
 **double** camY = **radiusConst** \* Math.*cos*(Math.*toRadians*(phi));

**return new** Vector(camX, camY, camZ).add(**lookAt**);  
}

Фигура 3.10.4

## Вмъкване и съхранение на обекти

Тази функционалност е изключително ограничена поради невъзможността да се достъпват данните на геометрията във ViroCore. Сегашната имплементация е временна. Когато ViroCore бъде актуализиран тя ще бъде променена изцяло.

Вмъкването става подобно на създаването на примитивен обект. Разликата е, че тук Uri обектът на файла трябва да бъде извлечен по някакъв начин в зависимост от избрания от потребителят файл. По същия начин както бе имплементиран избора на текстура е направен и избора на файл на модел. Uri инстанцията, която получаваме обаче, е от тип "content://". За да може даден модел да бъде зареден от loadModel функцията на Viro е нужно Uri от тип "file://". Един трик за преминаване от единия тип в другия е да се отвори файл с пътя на "content://" Uri обекта. След това да се състави ново Uri от този файл използвайки Uri.fromFile() функцията.

Uri **fileUri** = Uri.*fromFile*(**new** File(  
 uri.getLastPathSegment().substring(4)));

Фигура 3.11.1

Самото четене на файла, за да може да се подаде текстът му на OBJObject класа, може да се направи и със "content://" Uri. Отваря се входен поток, използвайки ContentResolver, след която той се превръща в четец. С него файлът може да бъде изчетен ред по ред и записан в StringBuilder.

InputStream **is** = getActivity().getContentResolver().openInputStream(uri);  
**assert** is != **null**;  
BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  
String line;  
**while** ((line = br.readLine()) != **null**) {  
 text.append(line);  
 text.append(**'\n'**);  
}  
br.close();

Фигура 3.11.2

Въпреки че няма достъп до данните на модела, благодарение на начина по-който се създават обектите (3.3), техните начални стойности могат да бъдат достъпени. Тъй като единични върхове не могат да се редактират, следва текущите позиции да са равни на начални ако приложим към тях трансформацията на обекта. Индексите за създаване на лица не се променят. Именно за това не се запазват като стойности при вмъкване а като текст.

За да се създаде правилно .OBJ представянето на даден обект се започва от името му, с първи символ на реда "о ". Така се разбира, че започва описанието на нов обект. След това се итерира през старите стойности на върхове и нормали, пресмятат се новите чрез трансформацията на обекта и се записват в нужният формат. Координатите на текстурата не се променят затова просто се използват началните. Накрая се добавят и индексите за лица и името на материала, който използва обектът. Това се повтаря за всеки един обект в сцената.

OBJObject **obj** = (OBJObject) n;  
output.append(**"o "**).append(obj.getName()).append(ln);  
**for**(Vector v: obj.getVertices()){  
 String vectorText = transformMatrix.multiply(v)  
 .toString().replace(**","**, **""**);  
 output.append(**"v "**).append(vectorText.substring(1, vectorText.length()-1)).append(ln);  
}  
*//repeat for normals***for**(Vector vt: obj.getTextureCoords()){  
 output.append(**"vt "**).append(vt.toString()).append(ln);  
}  
output.append(**"usemtl "**).append(obj.getMaterials().get(0).getName());  
output.append(**"s off"**).append(ln);  
output.append(obj.getFacesText());  
output.append(ln);

Фигура 3.11.3

Освен самите обекти, при запазване в .obj формат, трябва да се запазят и материали. Те се описват във файл със същото име но с окончание .mtl. Текстът на този файл се констуира по подобен начин като този на .obj. След като и двата текста бъдат съставени те трябва да бъдат запазени на устройството. Това се прави чрез exportFile() функцията в TopMenuFragment.

File **file** = **new** File(Environment.getExternalStoragePublicDirectory(  
 Environment.DIRECTORY\_DOCUMENTS), filename);  
  
FileOutputStream **outputStream** = **null**;  
**try** {  
 file.createNewFile();  
 outputStream = **new** FileOutputStream(file, **true**);  
  
 outputStream.write(body.getBytes());  
 outputStream.flush();  
 outputStream.close();

Фигура 3.11.4

Тя създава нов файл в папка "Documents" на вградената памет на устройството. Ако няма такава, тя ще се създаде автоматично. След коeто, използвайки FileOutputStream записва тялото на текстът в този файл. Накрая този запис се запазва и потокът бива затворен.

# Четвърта Глава

# Изисквания. Ръководство на потребителя

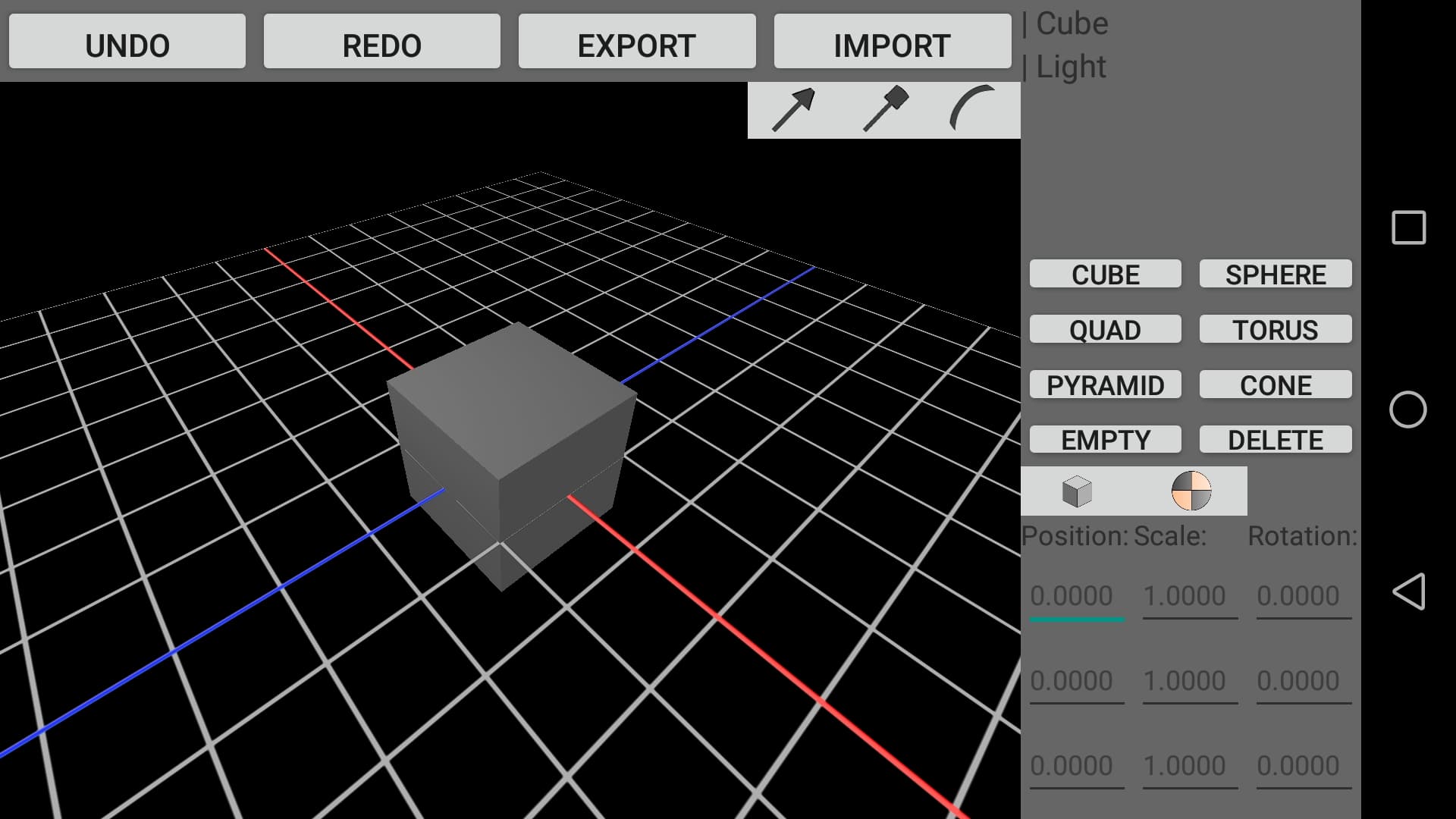
## Изисквания

Единственото изискване за стартиране на приложението е наличието на Android устройство с минимална версия 8.0 Lolipop (Api level 26). Това е необходимо за нормалната работа на ViroCore.

Желателно е устройството да разполага и с по-голям екран защото това би улеснило работата с приложението.

## Инструкции за ползване

При стартиране на приложението потребителят ще види екран подобен на *Фигура 4.2.1* (винаги ориентиран хоризонтално). Той се състои от главен изглед, на който се вижда сцената, и множество потребителски интерфейс около него. В тази глава е обяснено как се използва то.



Фигура 4.2.1

### Главен изглед

Главния изглед заема по-голямата част от екрана. На него потребителят може да вижда създадената сцена и да навигира по нея в триизмерното пространство. В началото тя представлява празен свят с един куб по средата. Също така има и координатна мрежа лежаща върху xy равнината, служеща за ориентир.

#### Контрол на камерата

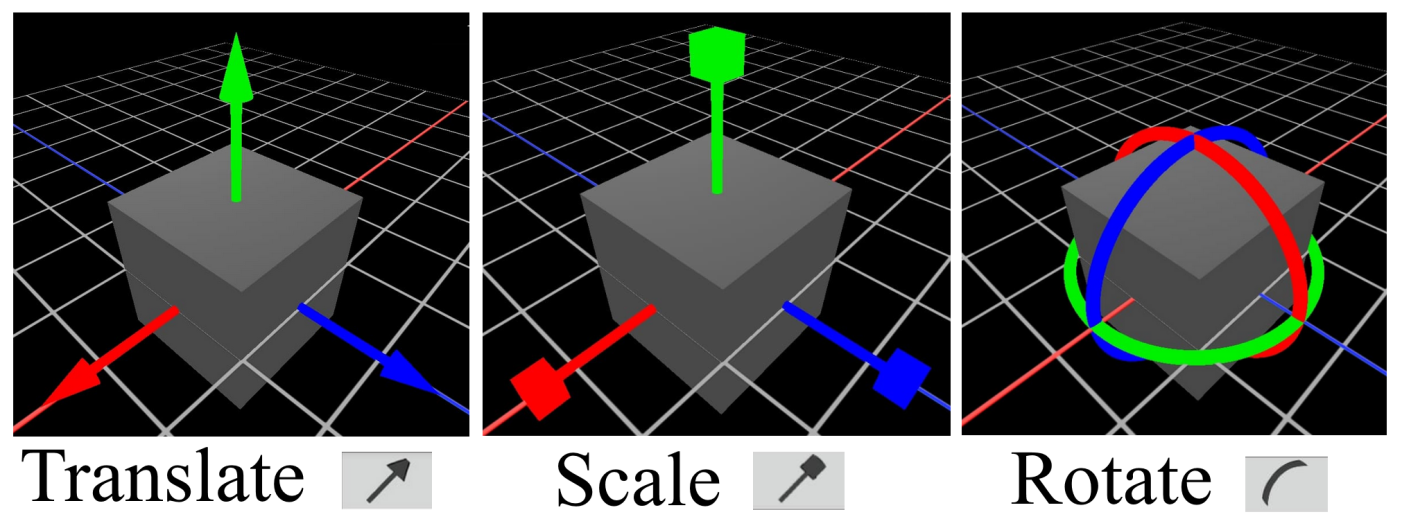
За да се завърти камерата е нужно единствено плъзгане с пръст по сцената. Тя се върти около една фокусна точка, която в началото е центъра на координатната система.

За приближаване и отдалечаване е нужно свиващо движение с двата пръста. Когато те се стесняват камерата се приближава към фокусната си точка, а когато се раздалечават тя се приближава.

За преместване на камерата настрани (паралакс ефект) , потребителят трябва да направи движение/плъзгане в една посока с два пръста.

#### Селектиране и манипулиране на обекти

Освен да разглежда, потребителят може и да взаимодейства с обектите в сцената през главния изглед. Когато натисне върху даден обект той бива селектиран. Върху него се появяват три стрелки перпендикулярни една на друга (по една за всяка ос). Червената е успоредна на оста x, зелената на y, синята на z. Има три вида помощни стрелки - за транслиране, оразмеряване и завъртане. За смяна на текущия вид стрелки служат бутоните в горния десен ъгъл на главния изглед. *Фигура 4.2.1.2.1* показва трите вида и съответния бутон служещ за превключването към тях.



Фигура 4.2.1.2.1

Стрелките за транслация изглеждат като обикновени стрелки. Всяка от тях служи за промяна на позицията на даден обект по дадената ос и може да се движи само по нея. За да направи това, потребителят трябва просто да докосне една от стрелките и да я довлачи до новата и позиция. Селектираният обект ще се придвижи заедно с нея.

Стрелките за скалиране ще оразмерят обекта по оста, за която отговарят. При издърпване на стрелката към обекта той ще се смали, при отдалечаване ще се уголеми.

Стрелките за ротация, които реално са окръжности, трябва да бъдат завъртяни с един пръст по равнината, в която лежат. Обектът ще се ротира заедно с движението на пръста.

### Меню за създаване на примитивни обекти



Фигура 4.2.2.1

По средата на дясната част на екрана се намира менюто за създаване на примитивни обекти *Фигура 4.2.2.1*. Потребителят може да го използва за да добавя нови, прости обекти към сцената. То се състои от няколко бутона подреди в две колони един върху друг вертикално. Поред от ляво на дясно и от горе на долу те служат за - създаване на куб, на сфера, на квадратен полигон, на пояс, на пирамида, на конус и на празен обект. Последният бутон служи за изтриване на селектирания в момента обект.

При създаване на обект неговата начална позиция е винаги центъра на координатната система без промяна трансформацията. Също така той използва материала по подразбиране.

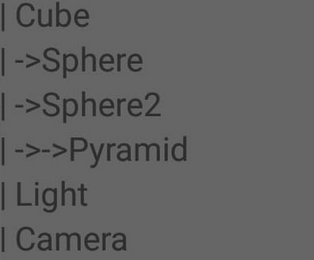
### Меню за промяна на параметрите на обекта

Намира се в долния десен ъгъл на екрана. Състои се от три колони с по три текстови полета. От ляво на дясно колоните отговарят за позиция, размер и ротация. Тези параметри се състоят от три числа представляващи трите оси. За да промени стойността на някой от тях, потребителят трябва да натисне някое от текстовите полета. От горе на долу те променят параметъра съответно по x, y и z оста. Например, ако потребителят въведе числото '1.5', в първото поле от първата колона, обектът ще промени позицията си по оста x на '1.5'.

За да може тази промяна да има ефект трябва преди това да има селектиран обект. върху който да се извърши.

### Изглед на йерархията

Изгледът показващ йерархията се намира се в горната дясна част на екрана. Той показва всички обекти в сцената както и връзките между тях. *Фигура 4.2.4.1* показва примерна подредба на йерархията. Разбираме, че Sphere и Sphere2 са дъщерни обекти на Cube, а Pyramid на Sphere2.



Фигура 4.2.4.1

Потребителят може да го използва за селектиране на обектите и за промяна на връзките между тях. Бързо натискане върху някой от обектите в йерархията е равносилно на натискане върху обекта в самата сцена. Той ще бъде селектиран по същия начин описан в точка 4.2.1.2.

Ако потребителят задържи пръста си върху някой от обектите, ще се появи негова "сянка". Тя може да бъде влачена по екрана. При отпускане на пръста съответния обект ще стане дъщерен на този, до когото е бил довлачен. Йерархията ще се обнови, така че да отразява правилно новата връзка. Ако няма такъв (вдигането на пръст е станало в част от екрана където няма член на йерархията), то сянката просто ще изчезне без промяна в позицията на обекта.

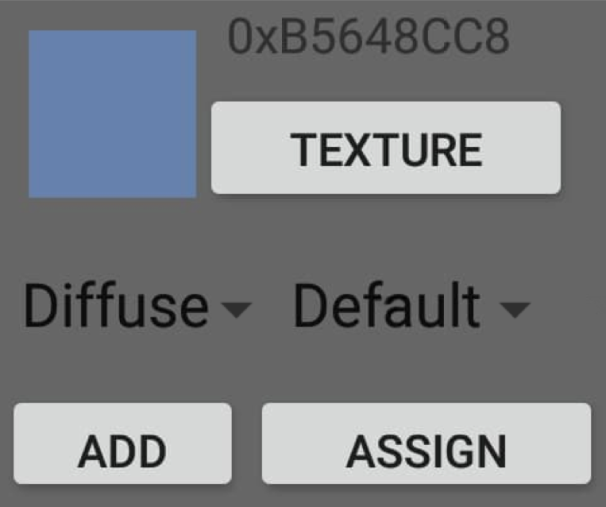
Щом един обект стане дъщерен, неговите трансформации ще се влияят от тези на обекта родител. Ако той бъде поместен, дъщерния обект ще се премести заедно с него. При завъртане на той също ще завърти но не по собствената си ос а по тази на бащиния обект. Трябва да се има предвид, че това ще завърти и цялата локална координатна система на обектa. Което значи, че ако потребителят промени параметъра x на дъщерния обект от менюто за промяна на основни параметри, неговият x ще се промени спрямо родителя си, а не спрямо основната координатна система на света, т.е. би изглеждало, че се движи накриво. Помощните стелки, обаче, биха променили позицията на обекта както нормално, тъй като те винаги са успоредни на осите на сцената.

### Меню за промяна на материала на обект

При стартиране на програмата изгледът на това меню не се вижда. То споделя местоположението си с менюто показващо основните параметри на обекта. За да бъде направено то видимо, се използват трите бутона намиращи се точно над него .

Чрез тях се превключва кое меню е активното на мястото на основните параметри (долния десен ъгъл). За активиране на менюто за материали е нужно да бъде натиснат средният бутон. Левият активира менюто с основни параметри.

При превключване към менюто за материали в долния десен ъгъл ще се появи следният изглед:



Фигура 4.2.5.1

Преди да променя материали е добре потребителят да е запознат как те се създават и прилагат към моделите. Падащото меню, което показва "Default", показва всички създадени материали. Чрез него се избира кой материал да бъде редактиран. В началото има само един, обикновен дифузен материал с бял цвят, който бива използван от куба в центъра на сцената, както и от новосъздадени примитивни обекти. Ако потребителят го промени, всички обекти, които създаде след това ще използват променения материал. За създаване на нов материал се натиска бутона "ADD". Потребителят ще трябва да въведе име, след което той ще бъде селектиран за редактиране. В началото той е същият като материала по подразбиране. Новосъздаденият материал може да бъде приложен към даден обект чрез бутона "ASSIGN". Преди това обекта трябва да бъде селектиран.

За промяна на цвета се натиска малкото квадратче показващо текущия цвят или текстът от дясно на него, който представлява цвета в шестнадесетичен вид. На екрана ще се появи следният прозорец:



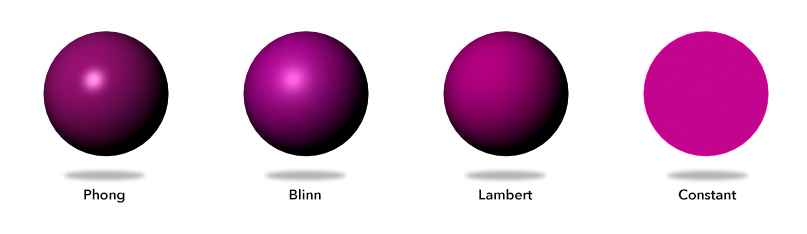
Фигура 4.2.5.2 [[[6]](#footnote-7)]

Големият цветен кръг се използва за избор на цвят, а долните два плъзгача за яркост и прозрачност. След като бъдат нагласени желаните стойности цветът се задава чрез бутона "Choose" в горния десен ъгъл. Ако потребителят не е доволен с резултата може да не избира цвета, а да затвори прозореца с бутона "Cancel" в горния ляв ъгъл.

Освен обикновен цвят, на даден материал може да му бъде зададена текстура. Това става чрез на натискане на бутона "Texture". Ще се появи нов екран, показващ файловете на устройството. Потребителят може да избере снимка, която ще бъде зададена като текстура на текущия материал.

Последното нещо, което може да бъде променяно е моделът на осветяване или как реагира обекта на светлина. Има четири възможност *Фигура 4.2.6.4*, които се избират от падащото меню до това за избор на материал.

* Diffuse - обекта отблъсква светлина еднакво във всички посоки.
* Phong - отблъсква светлина предимно в една посока. Изглежда сякаш има отражение на нея върху обекта подобно на метален лъскав материал.
* Blinn - подобно на specular но с по-разсеяно отражение.
* Constant - обекта не реагира на светлина. За разлика от дифузни обекти тук дори няма и сенки. Обекта е оцветен с един и същи цвят навсякъде, от където и да го погледнеш.



*Фигура 4.2.5.3* [[[7]](#footnote-8)]

### Горно меню

То представлява тънка лента от бутони в горната част на екрана. Стандартно е в такива менюта да се слагат настройки на програмата, опции за запазване и т.н. Тук има четири бутона.

Два от тях са за undo и redo. Те са много важни за всеки един редактор. Ако потребителят направи грешка, може да се върне назад чрез undo и да я предотврати. Ако размисли, може отново да нанесе промяната с redo. Единствено смислени промени върху обектите биват третирани като действия, които могат да бъдат върнати назад. Например ако потребителят премести камерата, през която гледа главния изглед, няма да може да се върне на предишната позиция.

Освен тях има и два бутона за вмъкване и съхранение. При натискане на бутона import/вмъкване, потребителят ще види пред себе си екран със файловете на устройството си. От там може да избере някой 3D модел съхранен в някой от поддържаните формати. Той ще бъде вкаран в сцената в центъра на света - точка с координати (0,0,0).

При натискане на export/съхранение на екрана ще се появи прозорец с вариантите за съхранение. За момента те са запазване като снимка и като OBJ формат. При избиране на OBJ потребителят ще бъде попитан да въведе име за файла. След това ще може да си го намери в папка "Documents" във вградената памет на устройството си заедно със съответния .mtl файл. Mtl файлът ще съдържа в себе си всички създадени материали дори и да не се използват от някой обект. Ако някой материал използва текстурам, тя трябва да бъде преместена в същатата папка като .obj и .mtl файловете, за да работи правилно при вмъкване в друга програма.

При избор на запазване като снимка, приложението ще направи екранна снимка само на изгледа със сцената. На тази снимка няма да се виждат ориентировачната решетка и стрелки, появили се при селектиране. За да е възможно запазване под каквато и да е форма, потребителят трябва да е предоставил достъп на приложението до паметта на устройството. Това не винаги става автоматично. В такъв случай, е нужно тези права да му бъдат дадени през настройките на телефона.

# Заключение

Този проект бе първото приложение, което съм правил за Android. Въпреки това, смятам че успях да се справя със заданието и да науча практиките в Android света.. Изискванията в по-голямата си част са спазени. Текущата версия на ViroCore сравнително ограничава дадени функционалности. Най-вече вмъкване и съхранение на обекти. ViroCore често получава актуализации, което ще позволи и развитие на дипломната работа.

В бъдеще могат да се добавят съхранение към повече формати и редактиране на отделни върхове от геометрията на обектите, тъй като сегашните вариант за моделиране включват единствено транслация, скалиране и ротация. Анимирането също е функционалност, която би могла да бъде добавена дори и със сегашната версия на Viro.

# Използвана литература

[1] Anupam Chugh. Android Fragment Lifecycle. <https://www.journaldev.com/9266/android-fragment-lifecycle> (17.01.2019)

[2] Raj Advani. ViroCore access vertices. 01.05.2018. <https://github.com/viromedia/virocore/issues/105> (15.12.2018)

[3][6] Hong Duan. Color Picker. 15.06.2018. [*https://android-arsenal.com/details/1/7068*](https://android-arsenal.com/details/1/7068) (10.01.2019)

[4] Daniel Thian. ViroCore Orbit Camera. 26.02.2018. <https://github.com/viromedia/virocore/issues/30> (20.11.2018)

[5] Almeros. Android Multitouch Gesture Detectors. 08.09.2012. <https://code.almeros.com/android-multitouch-gesture-detectors/> (20.02.2019)

[7] ViroCore Lightning and Materials [*https://virocore.viromedia.com/docs/3d-scene-lighting*](https://virocore.viromedia.com/docs/3d-scene-lighting) (02.02.2019)

[8] ViroCore documentation<https://virocore.viromedia.com/docs>

[9] Android developer guide <https://developer.android.com/guide>

GitHub repository - <https://github.com/toniuyt123/Quire3D>

1. [1] Anupam Chugh. Android Fragment Lifecycle. <https://www.journaldev.com/9266/android-fragment-lifecycle> (17.01.2019) [↑](#footnote-ref-2)
2. [2] Raj Advani. ViroCore access vertices. 01.05.2018. <https://github.com/viromedia/virocore/issues/105> (15.12.2018) [↑](#footnote-ref-3)
3. [3] Hong Duan. Color Picker. 15.06.2018. [*https://android-arsenal.com/details/1/7068*](https://android-arsenal.com/details/1/7068) (10.01.2019) [↑](#footnote-ref-4)
4. [4] Daniel Thian. ViroCore Orbit Camera. 26.02.2018. <https://github.com/viromedia/virocore/issues/30> (20.11.2018) [↑](#footnote-ref-5)
5. [5] Almeros. Android Multitouch Gesture Detectors. 08.09.2012. <https://code.almeros.com/android-multitouch-gesture-detectors/> (20.02.2019) [↑](#footnote-ref-6)
6. [6] Hong Duan. Color Picker. 15.06.2018. [*https://android-arsenal.com/details/1/7068*](https://android-arsenal.com/details/1/7068) (10.01.2019) [↑](#footnote-ref-7)
7. [7] ViroCore. Lightning and Materials. [*https://virocore.viromedia.com/docs/3d-scene-lighting*](https://virocore.viromedia.com/docs/3d-scene-lighting) (02.02.2019) [↑](#footnote-ref-8)