SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dejvid Topler

3D GRAFIKA NA MOBILNIM PLATFORMAMA

DIPLOMSKI RAD

Varaždin, 2013.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dejvid Topler

Matični broj: 39562/10–R

Studij: ***Informacijsko i programsko inženjerstvo***

3D GRAFIKA NA MOBILNIM PLATFORMAMA

DIPLOMSKI RAD

Mentor:

Dr. sc. Ivan Hip, docent

Varaždin, rujan 2013.

Sadržaj

1. Uvod

Zadnjih nekoliko godina u informatičkom svijetu zasigurno pripada razvoju mobilnih platformi, koje su naglo zaživjele, te su postale svakodnevica u životu svakog pojedinca. U nizu događaja i preobrata na tržištu, Android platforma se istaknula kao jedna od vodećih platformi u području mobilnih uređaja.

Napredak u svijetu mobilnih uređaja može se uočiti na dnevnoj razini. Uslijed brzog razvoja hardvera došlo je i do potrebe za kompleksnijim i zahtjevnijim aplikacijama, a time i do razvoja alata koji omogućuju kreiranje takvih aplikacija. Jedan od takvih alata je OpenGL ES.

OpenGL ES 2.0 je standard za razvoj napredne računalne grafike, prilagođen za mobilne platforme. Standard je nastao iz potrebe da se robusni sustavi za razvoj računalne grafike minimiziraju i prilagode ograničenjima mobilnih platformi.

Cilj ovog rada je napraviti programsko rješenje za prikaz trodimenzionalnog modela varaždinskog glavnog trga (Trg kralja Tomislava - Korzo) na Android platformi, koristeći standard OpenGL ES 2.0. S obzirom da je ovo relativno novi standard, ne postoji mnogo izvora i primjera za kreiranje računalne grafike koristeći ovaj standard. To je dodatna motivacija i cilj da se rad koncipira da bude naputak, odnosno niz koraka, za izradu aplikacija koristeći OpenGL ES 2.0 i Android platformu. Ovim radom će biti prikazani i objašnjeni elementi za rad s aplikacijskim programskih sučeljem OpenGL ES-a, osnove rada s programima za sjenčanje i prikaz razvoja Android aplikacije.

Rad, uz Uvod i Zaključak, sadrži četiri glavna poglavlja. U poglavlju OpenGL ES je predstavljen OpenGL ES 2.0 standard. Opisan je način kako OpenGL ES funkcionira i objašnjeni su osnovni pojmovi koji su potrebni u radu sa aplikacijskim programskim sučeljem tog standarda. U trećem poglavlju opisana je Android platforma i arhitektura sustava. Prikazano je kako započeti s razvojem Android aplikacije i koncepti na kojima se razvoj takvih aplikacija temelji. U četvrtom poglavlju je prikaz programskog rješenja, s detaljima implementacije. U zadnjem, petom poglavlju je korisnička dokumentacija s opisom korištenja Android aplikacije.

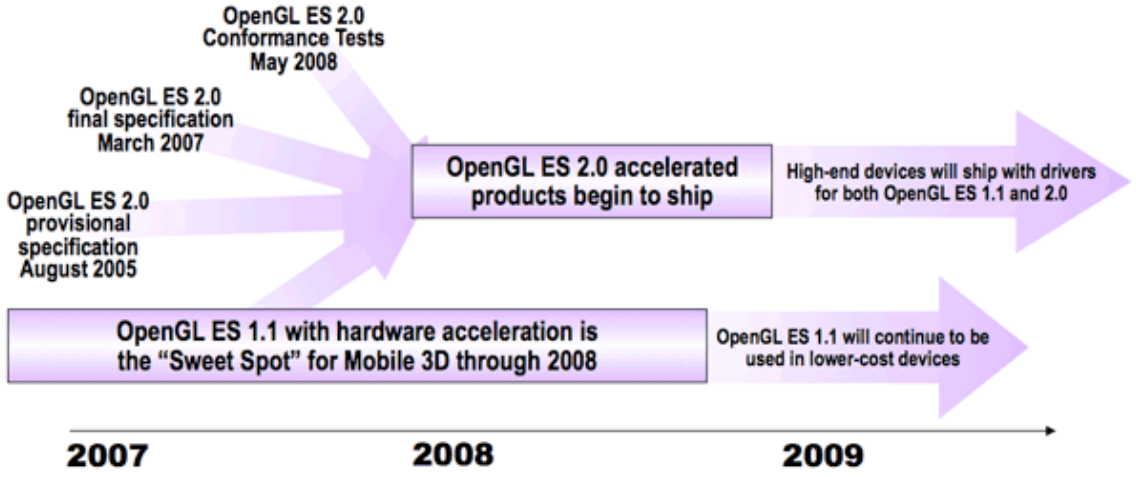
2. OpenGL ES

OpenGL ES je aplikacijsko programsko sučelje (engl. *application programming interface)* prilagođeno za rad na mobilnim i ugradbenim platformama. OpenGL ES je jedan od standarda Khronos groupe, neprofitne organizacije koja se bavi razvojem standarda za aplikacijska programska sučelja za 3D grafike. Osim OpenGL ES-a, Khronos groupa je razvila sljedeće otvorene standarde [Khronos Group, 2013a]:

- OpenGL
- WebGL
- OpenMax
- Collada
- WebCL
- OpenSL
- i dr.

OpenGL ES je podskup OpenGL aplikacijskog programskog sučelja, koje se koristi za razvoj napredne 3D grafike na Linux, Unix, Mac OS i Windows platformama, iz razloga što je OpenGL ES specifično namijenjen za rad na mobilnim platformama. Mobilne platforme, u odnosnu na desktop platforme su značajno uskraćene po pitanju performansi samih uređaja. Osim procesorske snage i memorije uređaja, postoje i druga ograničenja poput veličine ekrana ili trajanja baterije, te su sve to razlozi da se za mobilne platforme razvio podskup alata koji će biti prilagođeni takvom okruženju. [Ginsberg, Munshi, Shreiner, 2008, str 1]

Postoji nekoliko verzija OpenGL ES-a izdanih od Khronos grupe: OpenGL ES 1.0, OpenGL ES 1.1, OpenGL ES 2.0 i OpenGL ES 3.0. Svaka od novijih verzija je donijela nešto naprednije i grafički zahtjevnije sukladno s razvojem samih uređaja. Na slici 1. je prikazan vremenski slijed razvoja pojedinih verzija OpengGL ES-a. Treba napomenuti da razvoj OpenGL ES-a napreduje u dvije grane, odnosno u verzije 1.x i 2.x. Ideja je bila da se pokrije što širi spektar uređaja, pa se tako verzijama 1.x nastoji omogućiti razvoj za uređaje slabijih performansi, dok se verzijom 2.x cilja na uređaje koji podržavaju mnogo zahtjevniju grafiku.



Slika 1. Razvoj OpenGL ES [Khronos Group, 2013c]

Velika razlika između verzije 2.0 i 1.x je što je uveden programabilni grafički protočni sustav (engl. *graphic pipeline*) s mogućnošću da izvršava programe za sjenčanje koristeći *OpenGL ES Shading language.* Pri tome je bitno napomenuti da verzija 2.0 nije kompatibilna s nižim verzijama, odnosno uređaji koji imaju podršku za verziju 1.x, a nemaju za verziju 2.0 neće moći pokrenuti aplikaciju koja koristi verziju 2.0. Verzije 1.x su međusobno kompatibilne.

2.1. OpenGL ES 2.0

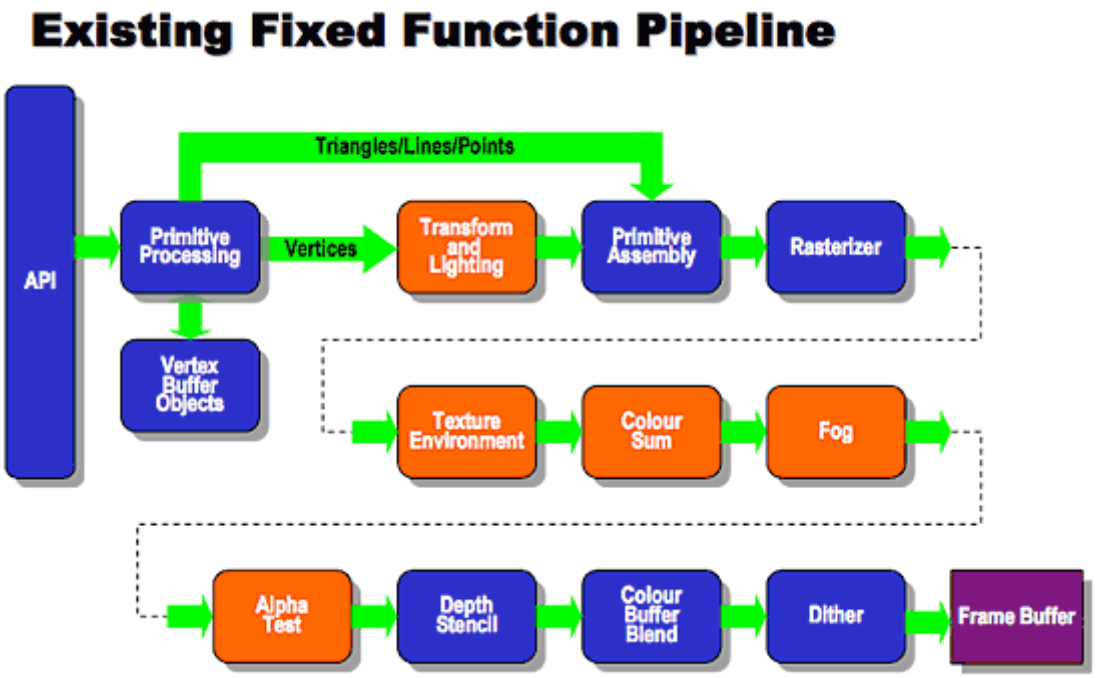
Za rješenje programskog problema izrade 3D modela varaždinskog glavnog trga (Trg kralja Tomislava, popularno Korzo) korišteno je sučelje OpenGL ES 2.0. OpenGL ES 2.0 je definiran kroz dvije specifikacije:

- *OpenGL ES 2.0 API* specifikacija koja definira način korištenja sučelja za rad s vrhovima, teksturama, spremnicima i slično.
- *OpenGL ES Shading Language Specification (OpenGL ES SL)* definira sva pravila i ograničenja vezana za korištenje programa za sjenčanje vrhova i točaka.

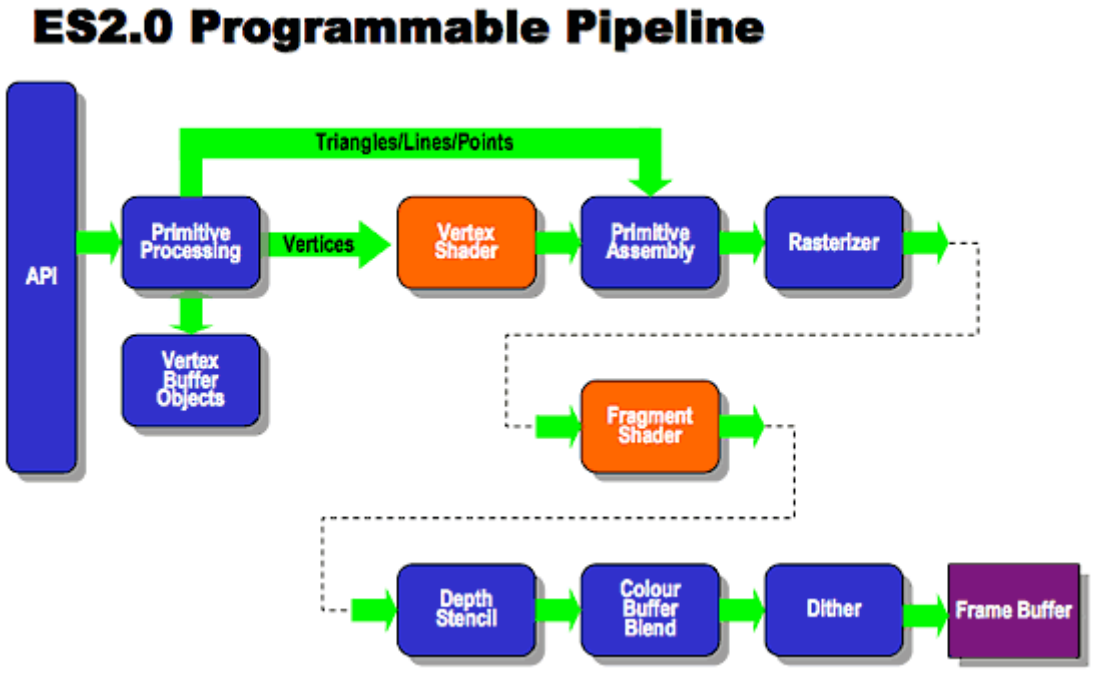
U narednim poglavljima će biti opisani dijelovi iz obje specifikacije koje je nužno poznavati kako bi se moglo raditi pomoću OpenGL ES 2.0 sučelja.

2.1.1. Grafički protočni sustavi OpenGL ES-a

Najveća razlika između OpenGL ES 1.x i 2.x verzija je u grafičkom protočnom sustavu. Verzije 1.x imaju fiksni grafički protočni sustav koji je prikazan na slici 2., dok verzije 2.x imaju programabilni grafički protočni sustav prikazan na slici 3. Na navedenim slikama narančastom bojom su oslikani elementi koji se razlikuju između jedne i druge verzije.



Slika 2. Fiksni grafički protočni sustav



Slika 3. Programabilan grafički protočni sustav [Khronos Group, 2013d]

Vidi se da verzije 1.x i 2.x imaju sličan grafički protočni sustav, s razlikama u procesoru vrhova (engl. *vertex shader*) i procesoru točaka (engl. *fragment shader).* Kako bi se shvatilo koju funkciju ti procesori imaju objasnit će se svaki korak u grafičkom protočnom sustavu:

- Međuspremnik vrhova (engl. *vertex buffer objects*) je vrsta međuspremnika koji sadrži koordinate vrhova objekata koji će se iscrtati. Međuspremnik se alocira u memoriji grafičke kartice, te su podaci u spremniku dostupni za sljedeće korake procesiranja.

- Procesiranje primitivnih objekata (engl. *primitive processing*) koristi međuspremnik vrhova te prosljeđuje sve potrebne informacije u procesor vrhova.

- Procesor vrhova (engl. *vertex shader*) može biti korišten za transformiranje pozicija vrhova, izračun formula za osvjetljenje za svaki vrh i generiranje koordinata tekstura. U programabilnom grafičkom protočnom sustavu odluka je na programeru što od tih funkcionalnosti želi koristiti, dok u fiksnom grafičkom protočnom sustavu na te transformacije nema direktnog utjecaja. U fiksnom grafičkom protočnom sustavu takve transformacije su implementirane u koraku „transformacije i osvjetljenje“ (engl. *transform and lightening*), te su one generalne i ne mogu se prilagoditi određenom scenariju. Procesor vrhova procesira jedan po jedan vrh koristeći program za sjenčanje vrhova. Detaljnije o programu za sjenčanje vrhova u poglavlju 2.5.1. [Simpson, 2006]

- Nakon što je procesor vrhova obradio vrhove modul za sastavljanje primitivnih objekata (engl. *primitive assembly*) izvodi obradu nad primitivnim objektima. Primitivni objekti su geometrijski likovi koji se koriste za iscrtavanje. Glavna zadaća ovog modula su operacije selektivno odbacivanje (engl. *culling*) i odrezivanje (engl. *clipping*). Pomoću metode odrezivanja određujemo da li je određeni primitivni objekt unutar ili izvan krnje piramide (engl. *frustrum*) koja predstavlja trodimenzionalni prostor koji će biti vidljiv na ekranu. Ako je objekt izvan krnje piramide objekt se odbacuje. U slučaju da je djelomično unutar krnje piramide tada se primitivni objekt odreže. Selektivno odbacivanje služi da se odbace primitivni objekti ovisno o tome na koju je stranu objekt okrenut, tj kako je orijentiran. [Ginsberg, Munshi, Shreiner, 2008, str 7]

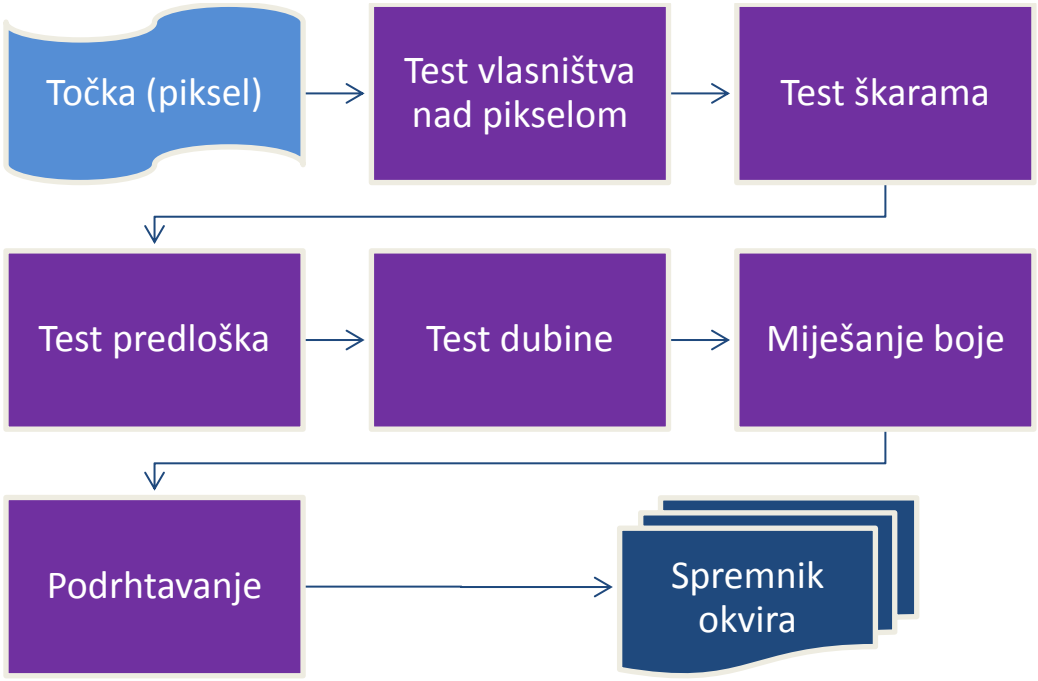
- Rasterizacijska jedinica (engl. r*asterizer unit*) služi da na temelju trodimenzionalnog objekta napravi dvodimenzionalnu sliku, na način da svaki primitivni objekt konvertira u skup podataka koji sadrže dvodimenzionalnu koordinatu, boju, te eventualno koordinate teksture i koji se naziva fragment (zapravo je to piksel, tj.

najmanji slikovni element). Fragment se prosljeđuje u procesor točaka, gdje se dodatno manipulira konačnom bojom pomoću programa za sjenčanje točaka. [Ginsberg, Munshi, Shreiner, 2008, str 7]

- Procesor točaka (engl. *fragment shader)* određuje konačnu boju svake točke. Procesor točaka koristi program za sjenčanje točaka za izračun konačne boje svakog fragmenta. Procesor točaka nema pristupa susjednim fragmentima, niti može mijenjati položaj fragmenta. Izračunate vrijednosti služe za ažuriranje stanja u spremniku slike(engl. *frame buffer*) ili eventualno za ažuriranje spremnika tekstura. Program za sjenčanje točaka također može odrediti konačnu boju točaka na temelju teksture koja je ispravno obrađena. Na slikama se može vidjeti da se u programabilnom protočnom sustavu koristi program za sjenčanje točaka, dok je u fiksnom grafičkom protočnom sustavu to malo drukčije. Program za sjenčanje točaka omogućuje, ali i obvezuje programera da sam implementira način na koji će se izračunati konačna boja za svaku točku. Detaljno o programu za sjenčanje točaka u poglavlju 2.5.2.

- Modul za operacije na fragmentima je na slikama 2. i 3. prikazan kao niz modula *Depth Stencil, Colour buffer Blend* i *Dither,* ali se puno više od toga procesira u ovom modulu. Na slici 4. je prikazan slijed testova koji se naprave na svakom fragmentu prije nego se postavi u spremnik fragmenata. To su sljedeći testovi:
    - **Test vlasništva nad pikselom** (engl. *Pixel ownership test*) služi da se odredi da li piksel na određenim koordinatama pripada OpenGL ES-u ili se uopće neće prikazivati (na primjer ako je prekriven drugim prozorom).
    - **Test škarama** (engl. *Scissor test*) odbacuje fragmente koji su izvan pravokutnika koji je definiran u prvom koraku ovog testa.
    - **Test predloška** (engl. *Stencil test*) služi kao maska, odnosno filtrira određena područja koja nisu vidljiva i fragment na takvim područjima odbacuje.
    - **Test dubine** (engl. *Depth test*) služi za odbacivanje fragmenata na istim pozicijama koji imaju veću dubinu, odnosno udaljeniji su od kamere.
    - **Miješanje boje** (engl. *Blending***)** kombinira boje novokreiranog fragmenta s fragmentom u spremniku slike.
    - **Podrhtavanje** (engl. *Dithering*)[1] minimizira broj različitih boja na temelju mogućnosti samog spremnika slike.

[1] Naziv metode koja se često koristi u obradi signala.



Slika 4. Modul za operacije na točkama

- Spremnik slike (engl. *frame buffer*) sadrži kolekciju slika koje će se iscrtati na ekran.

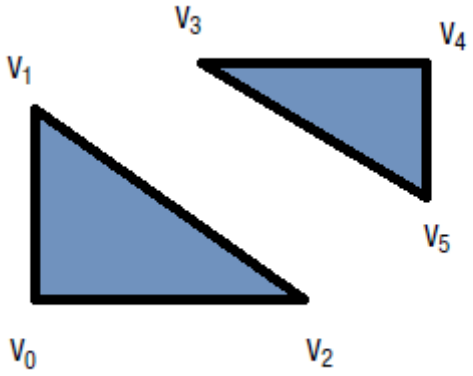
Na slikama nije prikazan još jedan element koji služi da se sadržaj spremnika slike prikaže na zaslonu uređaja, a to je EGL. EGL je zapravo sučelje između Khronos aplikacijskih sučelja, npr. OpenGL ES, i različitih platformi na kojima su ta aplikacijska sučelja podržana, npr. Android. EGL pruža mehanizam za kreiranje sučelja za iscrtavanje na koja korisnik sučelja može iscrtati sadržaj, kreira kontekst za korisnika i sinkronizira iscrtavanje, odnosno simulira rad nativne platforme. [Khronos Group, 2013e]

2.2. Primitivni objekti

Primitivni objekti su geometrijski likovi koji se koriste kao osnovni elementi za iscrtavanje kompleksnih objekata i odnose se na sljedeća tri tipa objekata: trokute, linije i točke.

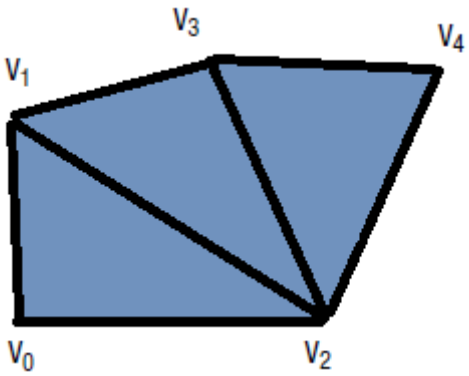
Trokut predstavlja geometrijski oblik koji je definiran s tri točke. *OpenGL ES* 2.0 ima podršku za sljedeće operacije s trokutima: GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN.

GL_TRIANGLES iscrta odvojene trokute, te se trokut definira s tri vrha. Na slici 5. je primjer iscrtavanja dva trokuta s definiranih šest vrhova.



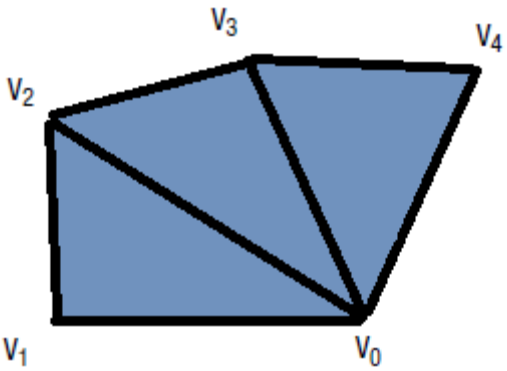
Slika 5. GL_TRIANGLE

GL_TRIANGLE_STRIP iscrta serije spojenih trokuta, na način da se definirani vrhovi od prethodnog trokuta ponovno koriste. Novi trokut u nizu se iscrta pomoću dva vrha prethodnog trokuta i uz pomoć još jednog vrha koji je definiran. Na slici 6. su iscrtana tri trokuta s definiranih pet vrhova. Prvi trokut koristi prva tri definirana vrha (v0, v1, v2), dok drugi ponovno koristi vrhove v1, v2, te se iscrta između v1, v2 i v3. Za treći trokut je potrebno definirati samo još jedan vrh v4.



Slika 6. GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN funkcionira slično kao GL_TRANGLE_STRIP osim što je prvi vrh svih trokuta uvijek prvi vrh prvog trokuta. Na Slika 7. GL_TRIANGLE_FAN 7. je prikazan niz od tri trokuta iscrtanih pomoću pet vrhova. Svi vrhovi imaju zajednički vrh v0, te se iscrtaju između vrha v0, zadnjeg vrha u prethodnom trokutu i jedinog novog vrha definiranog za taj trokut.



Slika 7. GL_TRIANGLE_FAN [Ginsberg, Munshi, Shreiner, 2008, str 128]

Linija predstavlja objekt koji je definiran s dvije točke, te također ima nekoliko podvarijanti: GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP.
GL_LINES iscrta niz odvojenih linija, te su potrebne po dva vrha za svaku liniju. GL_LINE_STRIP iscrta niz povezanih linija, gdje je završna točka prethodne linije, početna točka sljedeće linije.
GL_LINE_LOOP ima istu funkcionalnost kao i GL_LINE_STRIP s razlikom da se iscrta dodatna linija koja spaja zadnji vrh u nizu s prvim vrhom.

Točka je definirana s tri koordinate, te su u OpenGL ES–u može koristiti pomoću naredbe GL_POINTS.

Primitivne objekte je u OpenGL ES 2.0 moguće iscrtati koristeći metode *glDrawArrays* ili *glDrawElements*.

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count)
```

Metoda *glDrawArrays* iscrta primitivne objekte na temelju tri parametra. Prvi parametar tip *GLenum* je tip objekta (GL_TRANGLE, GL_LINE_STRIP ..), drugi parametar je indeks u listi vrhova od kojeg se počinje s iscrtavanjem, te je zadnji parametar ukupni broj indeksa koji se iscrtaju. Kako bi se iscrtala kocka pomoću GL_TRIANGLES poziv metode bi trebao izgledati ovako:

```
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Ovim pristupom je potrebno definirati 36 vrhova iz razloga što kocka ima šest stranica, a za svaku stranicu je potrebno po dva trokuta, dok se svaki trokut mora definirati s tri vrha.

Drugačiji pristup je korištenjem metode *glDrawElements.*

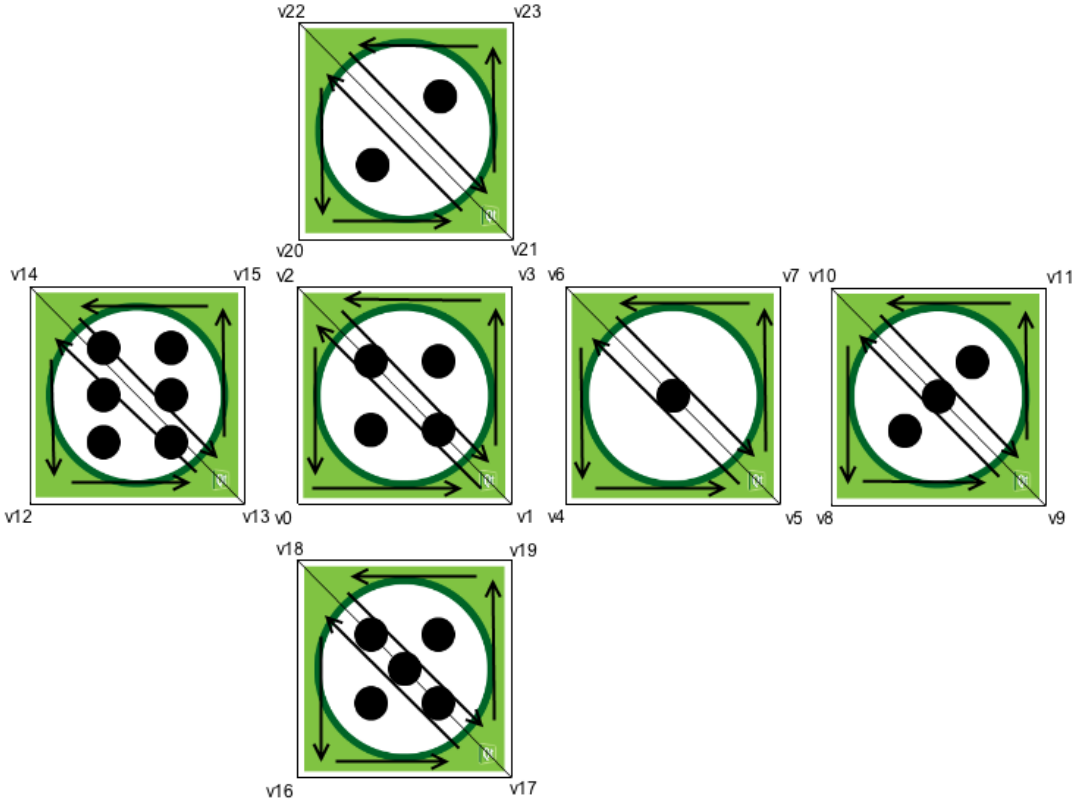
```
void glDrawElements(GLenum mode, GLsizei count,
GLenum type, const GLvoid *indices)
```

Prvi parametar je tip primitivnog objekta, drugi broj vrhova za iscrtati i treći parametar je tip indeksa spremljenih u međuspremniku indeksa. Međuspremnik indeksa predstavlja niz indeksa koji će se koristiti za iscrtavanje objekta, te je četvrti parametar referenca na taj međuspremnik.

```
GLfloat vertices[] = { … };// (x, y, z)  za svaki vrh kocke (osam vrhova)
GLubyte indices[36] = {
0, 1, 2, 0, 2, 3,
0, 3, 4, 0, 4, 5,
 0, 5, 6, 0, 6, 1,
 7, 6, 1, 7, 1, 2,
 7, 4, 5, 7, 5, 6,
 7, 2, 3, 7, 3, 4 };
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, indices);
```

Na primjeru iznad je prikazan poziv metode *glDrawElement* s parametrima za iscrtavanje kocke pomoću GL_TRIANGLES. Također je prikazan međuspremnik indeksa (varijabla *indices)* s 36 definiranih indeksa. Indeksi će omogućiti da se definiranih osam vrhova koriste prema pravilima definiranim kroz indekse. Npr. u primjeru iznad linija 3, predstavlja dio indeksa koji će definirati dva trokuta. Prvi trokut će imati vrhove definirane u *vertices* varijabli s indeksima 0, 1, i 2, a drugi trokut s indeksima 0, 2 i 3.

Gore navedeni primjer vrijedi u slučaju da za kocku nije potrebno imati adresiranu svaku stranicu. Međutim, to je potrebno u slučaju ako svaka stranica treba imati različite boje ili ako se žele koristiti teksture. U tom slučaju je potrebno definirati 24 vrha, odnosno četiri vrha za svaku stranicu. Na slici 8. je prikazano kako se vrhovi pridružuju pojedinim stranicama.



Slika 8. Primjer kocke [qt-project.org, 2013]

2.3. Buffer objects

U prethodnom poglavlju se govorilo o primitivnim objektima, o njihovim vrhovima i indeksima, te kako se ti objekti iscrtaju. U ovom poglavlju će biti objašnjeno kako definirati vrhove objektima, te kako te informacije učiniti dostupnima za daljnje procesiranje.

Informacije o vrhovima se mogu sastojati od koordinata samih točaka, normala i tekstura, te postoje dva uobičajena načina za spremanje tih informacija. Prvi način je spremanje svih informacija o vrhu slijedno u isti spremnik. Takav pristup se naziva lista struktura (engl. *array of structures).* Drugi način je spremanje različitih atributa u različite spremnike, npr. koordinate se spremaju u spremnik za koordinate, normale u spremnik za normale itd. Drugi pristup se naziva *structure of arrays*.

Informacije o vrhovima je potrebno proslijediti u procesor vrhova kako bi se nad njima odradile dodatne manipulacije, te na kraju kreirala konkretna slika u spremniku slika. Nakon što je konačna slika u spremniku slika, EGL se brine da se ista iscrta na zaslonu. Informacije

se prosljeđuju na način da se postave kao atributi u procesoru vrhova. Informacije o vrhovima se nalaze u kontekstu aplikacije, tj. iz aspekta OpenGL-a informacije su u klijentskoj aplikaciji. Kako bi informacije bile dostupne potrebno je koristiti *buffer object* da bi se informacije spremile u memoriju same grafičke kartice. Postoje dva tipa b*uffer object-a*, GL_ARRAY_BUFFER koji se koristi za kreiranje međuspremnika za informacije o vrhovima i GL_ELEMENT_ARRAY_BUFFER koji služi za kreiranje međuspremnika za indekse.

```
glGenBuffers(1, vbo, 0);
glBindBuffer(GLES20.GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GLES20.GL_ARRAY_BUFFER, vertexBuffer.capacity() *
BYTES_PER_FLOAT, vertexBuffer, GLES20.GL_STATIC_DRAW);
```

Na primjeru iznad je prikazano kako kreirati *Buffer object* koji će spremati informacije o vrhovima. Prva linija generira identifikator i sprema ga u listu *vbo* na poziciju 0. U drugoj liniji se odabire trenutni aktivni *Buffer object* s identifikatorom koji je prethodno generiran, te se prvim parametrom navodi da će ovaj spremnik služiti za informacije o vrhovima. Metoda *glBindBuffer* implicitno alocira memorijski prostor na grafičkoj kartici. U trećoj liniji trenutno aktivnom *Buffer object-u* pridružujemo referencu na spremnik koji sadrži informacije o vrhovima. Potrebno je napomenuti da se spremnik (u gornjem primjeru varijabla *vertexBuffer*) nalazi na klijentskoj strani.

2.4. Teksture

OpenGL ES 2.0 ima podršku za dva oblika tekstura: 2D teksture i *cube map* teksture. *Cube map* tekstura se sastoji od šest 2D tekstura koje zatvaraju kocku i najčešće se koristi za preslikavanje okoline (engl. *environment mapping).* 2D teksture su najčešći oblik tekstura koji se koriste, te sadrže dvodimenzionalni niz informacija o slici. Kada je riječ o 2D teksturama koordinata teksture je definirana prema indeksima, tj. par indeksa u listi koja sadrži informacije o 2D teksturama predstavljaju koordinate teksture. Koordinate su u normaliziranom obliku, s time da je donja lijeva strana ima koordinate (0.0, 0.0), dok gornja desna ima koordinate (1.0, 1.0). Vrijednosti izvan navedenog raspona su dozvoljene ali ponašanje je potrebno definirati pomoću *texture wrapping mode.* Postavljeni *texture wrapping mode* je GL_REPEAT, što će rezultirati multipliciranjem teksture za indeks definiran kroz koordinate. [Ginsberg, Munshi, Shreiner, 2008, str 181-186]

Kako bi se određena tekstura dodala na neki objekt potrebno je napraviti nekoliko koraka:

- **Generirati teksturu**

  Teksture se generiraju koristeći metodu prikazanu u sljedećem primjeru. Prva argument je broj tekstura koje treba generirati, a drugi referenca na listu u koju će se spremiti identifikatori za svaku od tekstura

```
void glGenTextures(GLsizei n, GLuint *textures)
```

- **Odabrati teksturu**

  Nakon što je tekstura generirana, preko identifikatora se može odabrati željena tekstura na kojoj je potrebno odraditi odgovarajuće operacije.

```
void glBindTexture(GLenum target, GLuint texture)
```

- **Učitati informacije**

  Svakoj kreiranoj teksturi se mogu učitati informacije o slici, najčešće kao *Bitmap* objekt. Metoda za učitavanje informacija je prikazana u sljedećem primjeru.

```
void glTexImage2D(…, const void* pixels)
```

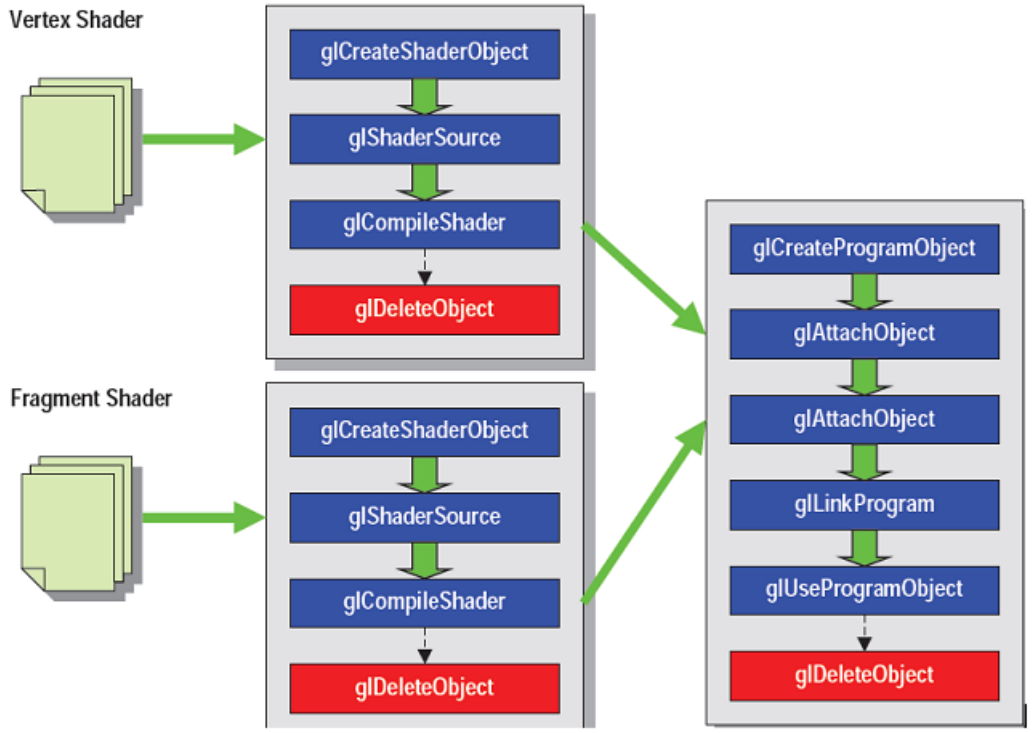
- **U programu za sjenčanje točaka pozvati metodu *texture2D***

  Metoda *texture2D* generira konačnu boju točke na temelju identifikatora teksture, te koordinata teksture. Rezultat metode *texture2D* je *vec4.*

2.5. Programi za sjenčanje

Programi za sjenčanje su obavezan dio programabilnog grafičkog protočnog sustava u arhitekturi OpenGL ES 2.0 sučelja. Jezik za sjenčanje se temelji na C-u i C++. C je proširen s novim tipovima kao *matrix* i *vector,* te su preuzete neke funkcionalnosti od C++. Postoje dvije vrste programa za sjenčanje, te je obavezne za implementirati obje kako bi se od trodimenzionalnog modela dobila dvodimenzionalna slika u spremniku slika. [opengl.org, 2013]

Na slici 8. je prikazan uobičajen slijed akcija koje se izvode s programima za sjenčanje.



Slika 9. Životni ciklus programa za sjenčanje [Multimedia Application Division, 2010]

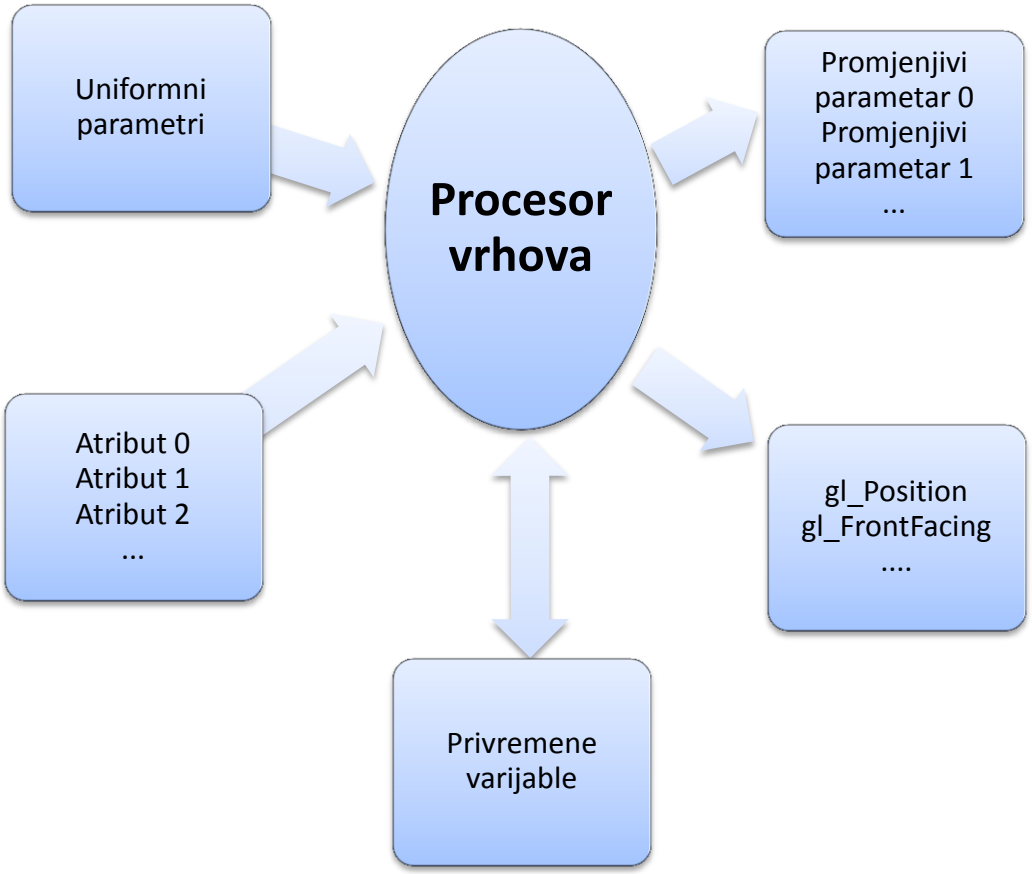
Prije nego se kreiraju programi za sjenčanje, potrebno je kreirati objekt *program* (engl. *program object*). Objekt *program* je kontejner za programe za sjenčanje preko kojeg se programi za sjenčanje mogu međusobno povezati. Prije nego se povežu potrebno ih je dodati u objekt *program*, a kako bi se to moglo potrebno je svaki od programa za sjenčanje kreirati i prevesti. OpenGL ES 2.0 API specifikacija definira kako se programi za sjenčanje učitavaju, povezuju, te kako im se prosljeđuju parametri.

2.5.1. Program za sjenčanje vrhova

Procesor vrhova je dio programabilnog grafičkog protočnog sustava, te koristi programe za sjenčanje vrhova za manipuliranje vrhovima. Procesoru vrhova je moguće proslijediti ulazne parametre, te ima izlazne parametre koji se prosljeđuju u procesor točaka. Na slici 5. su prikazani ulazi i izlazi u procesor vrhova. Ulazni podaci se sastoje od sljedećeg seta informacija:

- Atributi su informacije o svakom vrhu u obliku liste vrhova.
- Uniformni parametri (engl. *uniforms*) se odnose na konstante.
- Program za sjenčanje vrhova se odnosi na izvorni kod ili izvršnu datoteku koja opisuje operacije koje će izvršiti procesor vrhova

Izlazni podaci su promjenjivi parametri (engl. *varyings*) i nekoliko ugrađenih posebnih varijabli. Od posebnih varijabli najvažnija je *gl_Position* varijabla, koja predstavlja poziciju vrhova.



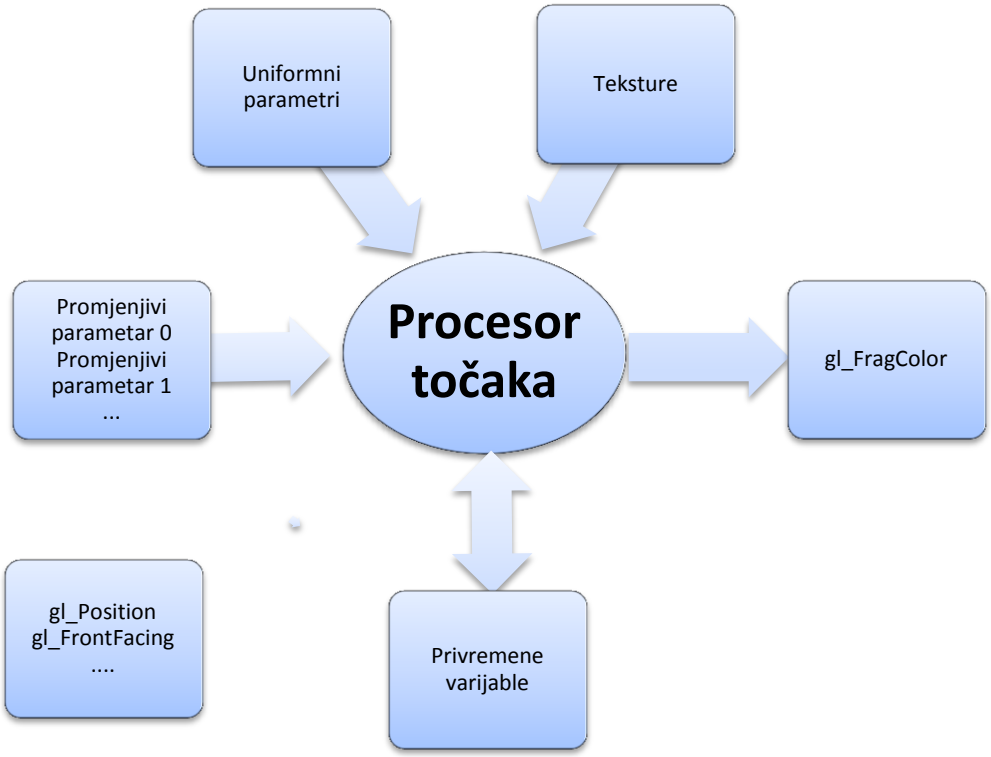
Slika 10. Procesor vrhova [Ginsberg, Munshi, Shreiner, 2008, str 149]

2.5.2. Program za sjenčanje točaka

Procesor točaka koristi program za sjenčanje točaka, kako bi procesirao točke, odnosno piksele. Na slici 6. je prikazan procesor točaka s svim ulazima i izlazima. Ulazne informacije su sljedeće:

- Promjenjivi parametri (engl. varying) su rezultati procesiranja procesora vrhova
- Uniformni parametri
- Teksture su slike koje predstavljaju teksture
- Program za sjenčanje točaka se odnosi na izvorni kod ili izvršnu datoteku koja opisuje operacije koje će izvršiti procesor točaka
- Specijalne varijable koje su rezultat procesora vrhova, poput *gl_Position*

Izlazna informacija je ugrađena varijabla *gl_FragColor* koja se koristi za prosljeđivanje boje točke u modul za sastavljanje primitivnih objekata.



Slika 11. Procesor točaka [Ginsberg, Munshi, Shreiner, 2008, str 219]

3. Android

Android je trenutno najpopularnija mobilna platforma na svijetu. Android pogoni više stotina milijuna mobilnih uređaja u više od 190 zemalja diljem svijeta. [developer.android.com, 2013a] Android platforma je razvijena 2003. od kompanije „Android“. Google je 2005. preuzeo Android platformu. Prva verzija izdana je 23. rujna 2008., od tada postiže sve veće uspjehe i bilježi stalni rast.

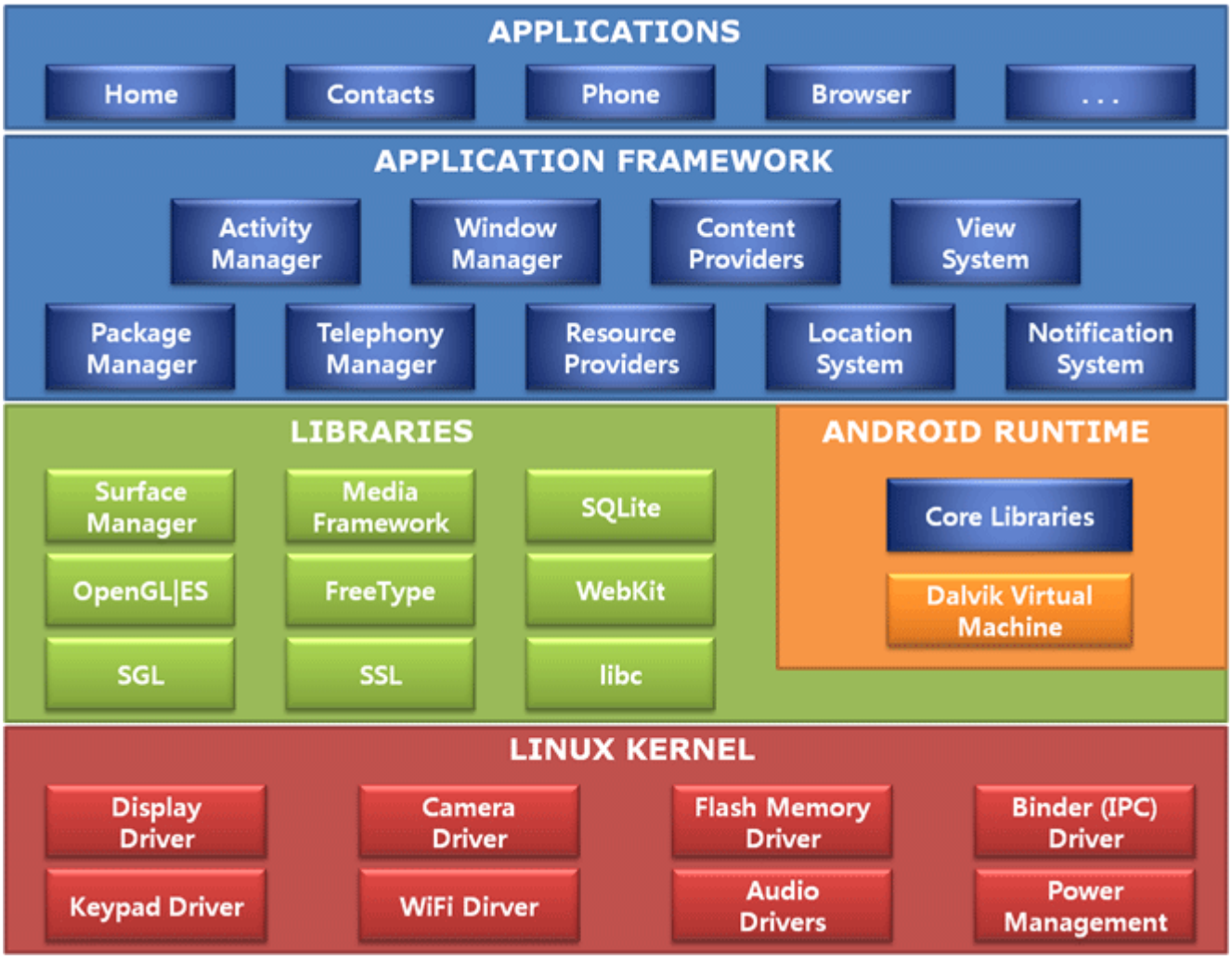
Slika 12. Vremenski prikaza rasta u broju uređaja [developer.android.com, 2013a]

Android platforma obuhvaća sam operacijski sustav, skup alata koji su integrirani u operacijskom sustavu, skup alata za razvoj aplikacija i platformu za distribuciju aplikacija. Alati za razvoj aplikacija su upakirani u Android SDK. Android SDK uključuje sučelje i niz alata potrebnih za kreiranje, testiranje i uklanjanje grešaka u aplikaciji. Kako pripremiti Android SDK za rad prikazano je u poglavlju 3.2.

3.1. Arhitektura

Android platforma omogućuje veliku fleksibilnost i brz razvoj aplikacija. Jedan od razloga je što je cijela arhitektura napisana u C/C++ jeziku, osim samih aplikacija koje su napisane u Java programskom jeziku. Android platforma je zasnovana na Linux jezgri, te uključuje aplikacijski okvir, operativni sustav, posebnu verziju *Java Virtual Machine* koja se naziva *Dalvik VM*, niz biblioteka za različitu primjenu i osnovne aplikacije. [satworks.blogspot.com, 2013].

Svi navedeni elementi su prikazani na slici 13.



Slika 13. Arhitektura android platforme [Dong Ho Han, 2013]

Android platforma je višekorisnički operacijski sustav gdje svaka aplikacija ima svoj korisnički račun, kojem su dodijeljena određena prava. Prava se dodjeljuju prema zahtjevima aplikacije i odnose se na prava pristupa određenim dijelovima sustava (npr. Internet pristup, pristup datotečnom sustavu i sl). U aplikaciji se mogu definirati koja prava su potrebna, te se samo ta prava dodjeljuju korisniku s kojim se aplikacija pokreće. Svaka aplikacija se pokreće u posebnom procesu, a svaki proces ima zasebnu virtualnu mašinu. [developers.android.com, 2013b]

3.1.1. Aplikacije

Android kao operacijski sustav nudi korisniku niz osnovnih aplikacija koje omogućuju osnovni rad s uređajem. Neke od aplikacija su:

- Klijent za e-poštu

- Aplikacija za upravljanje SMS porukama
- Kalendar
- Karte
- Web preglednik
- Aplikacija za upravljanje kontaktima

Sve navedene aplikacije su napisane u Java programskom jeziku.

3.1.2. Aplikacijski okvir

Aplikacijski okvir omogućuje pristup svim funkcionalnostima koje pruža Android platforma. Aplikacijski okvir ne ograničava programera po pitanju ponovnog korištenja koda, odnosno implementiranih funkcionalnosti. [satworks.blogspot.com, 2013]

3.1.3. Biblioteke

Android platforma koristi niz biblioteka napisanih u C/C++ programskom jeziku za rad sa naprednom grafikom, SQL bazama i sl. Svi navedeni moduli su programeru dostupni kroz aplikacijski okvir. [satworks.blogspot.com, 2013]

3.1.4. Android radno okruženje

Android radno okruženje se temelji na DVM (*Dalvik Virtual Machine*) koje se inicijalizira za svaku aplikaciju pokrenutu na Android platformi. DVM je virtualna mašina, veoma slična kao JVM (Java Virtual Machine), ali je prilagođena za rad na mobilnim platformama. Za razliku od JVM, DVM je prilagođena da radi u uvjetima s malom procesorskom snagom, s veoma malo radne memorije i u operacijskom sustavu bez *swap* memorije. DVM pakira međukod (engl. *byte code*) dobiven iz izvornog koda u *.dex* (engl. *Dalvik executable*, skraćeno *.dex*) datoteke, koje imaju sličnu funkcionalnost kao *.jar* datoteke u JVM. [Bornstain, 2013]

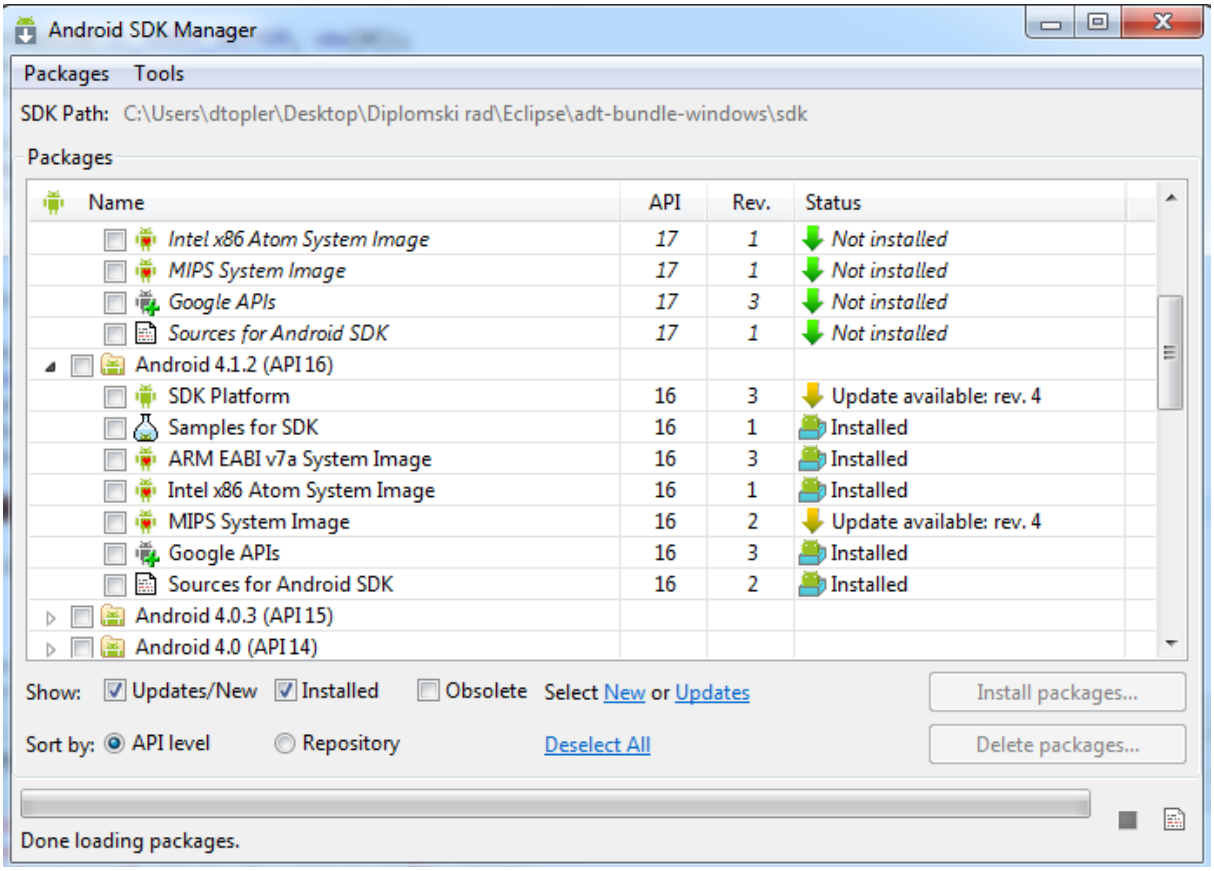
3.1.5. Linux jezgra

Android se oslanja na Linux verziju 2.6. za sustavne servise poput servisa za sigurnost, servisa za upravljanje memorijom, procesima, mrežom i pogonskim programima. [satworks.blogspot.com, 2013]

3.2. Android projekt

Android aplikacije se razvijaju u Java programskom jeziku. Da bi kreirali odgovarajući projekt, prvo je potrebno pripremiti radno okruženje.

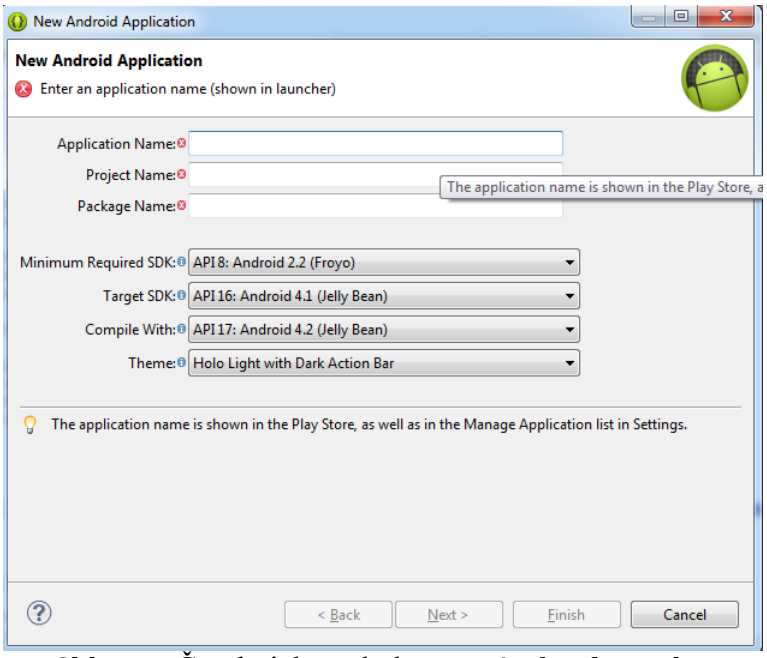
Za pripremu radnog okruženja dovoljno je preuzeti i raspakirati *Android ADT bundle*. *Android ADT bundle* sadrži *Eclipse* sa ugrađenim ADT (engl. A*ndroid Developer Tools*). Iz *Eclipse* okruženja moguće je otvoriti *SDK Manager* kao na slici 14.



Slika 14. SDK Menadžer

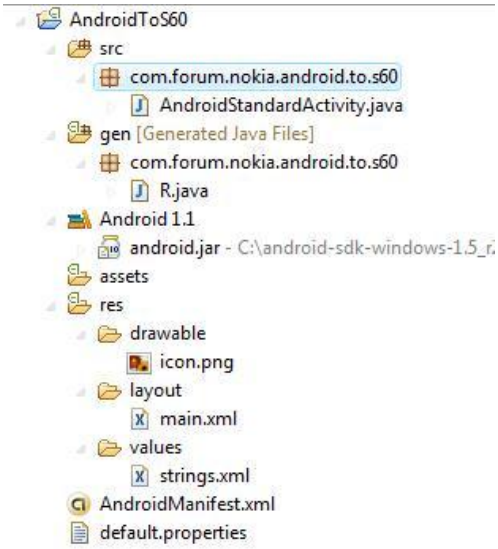
SDK menadžer služi za ažuriranje određenih verzija Android SDK. Nakon preuzimanja odgovarajuće SDK verzije, može se kreirati Android projekt. Android projekt se dodaje kroz čarobnjak prikazan na slici 15. Čarobnjak za dodavanje Android aplikacije se otvori odabirom opcije *File -> New -> Android Application Project*. Potrebno je popuniti sljedeće podatke:

- Naziv aplikacije
- Minimalnu verziju Android SDK
- Ciljanu verziju Android SDK (API 1.0 – API 17.0)
- Verziju Android SDK kojom će se aplikacija prevoditi – moguće odabrati *Google APIs* koji nudi dodatne biblioteke (npr. OpenGL ES)
- Izgled aplikacije (odabir nekog od predložaka)



Slika 15. Čarobnjak za dodavanje Android projekta

U narednim koracima čarobnjaka moguće je kreirati početnu aktivnost, ikonu i sl. Klikom na gumb *Finish* kreira se novi Android projekt. Na slici 16. je prikazan osnovni dio projekta koji je generiran na ovaj način.



Slika 16. Primjer android projekta [developer.nokia.com 2013]

Direktorij *src* sadrži pakete s klasama, odnosno sami izvorni kod aplikacije. U direktoriju *gen* se nalazi sadržaj koji se generira prilikom prevođenja projekta. *R.java* je generirana klasa koja sadrži reference na resurse te nije poželjno modificirati ovu datoteku. Direktorij *res* sadrži sve resurse koji se koriste u aplikaciji. U *drawable* direktoriju se nalaze slike koje se koriste kao ikone te ih je moguće definirati u više različitih veličina. Android omogućuje da se, ako postoje slike različitih dimenzija, implicitno odabiru one koje odgovaraju dimenzijama

uređaja. Npr. ako je riječ o tablet uređaju tada će se odabrati slika najvećih dimenzija. *Layout* direktorij sadrži XML opise aktivnosti, odnosno služi za definiranje izgleda samih aktivnosti. *Values* direktorij sadrži popis prijevoda koji se mogu koristiti kroz aplikaciju. Koristeći ovaj pristup moguće je napraviti višejezičnu aplikaciju. Na primjeru ispod je prikazano kako se definiraju prijevodi:

```
<string name="labelWaiting">Molimo pričekajte</string>
```

AndroidManifest.xml datoteka je obavezna za svaku Android aplikaciju te se nalazi u korijenskom direktoriju projekta. *AndroidManifest.xml* definira sljedeće neophodne informacije: [developer.android.com, 2013c]

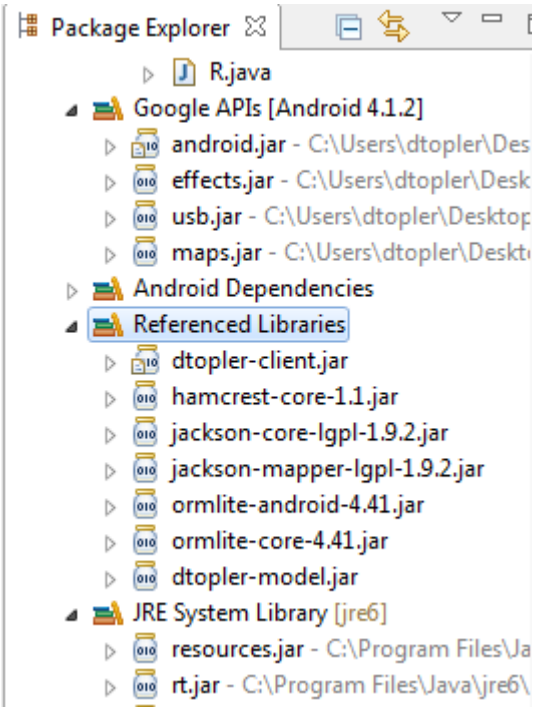
- Definira Java paket
- Opisuje komponente aplikacije (npr. aktivnosti), te definira koje klase implementiraju pojedinu komponentu
- Deklarira niz prava koje aplikacija zahtijeva
- Definira niz ovisnih biblioteka
- Definira minimalnu razinu Android API-a koju aplikacija zahtjeva

U okviru ispod je prikazan dio *AndroidManifest.xml* korišten u izradi programskog rješenja.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="org.dip.android"
   <uses-sdk android:minSdkVersion="14" />
   <uses-feature android:glEsVersion="0x00020000" android:required="true" />
   <uses-permission android:name="android.permission.INTERNET"/>
   <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORGE" />
   <application
    android:icon="@drawable/ic_launcher"
     android:label="@string/app_name"
     android:debuggable="true" >
     <activity android:name=".activity.TableOfContents"
        android:label="@string/app_name"
        android:icon="@drawable/toc" >
        <intent-filter>
           <action android:name="android.intent.action.MAIN" />
```

```
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".activity.OpenGLActivity"
        android:label="@string/labelOpenGl" />
  </application>
</manifest>
```

U pregledu na slici 16. nije omogućen prikaz još jednog bitnog elementa, a to su biblioteke koje su prikazane na slici 17.



Slika 17. Android projekt – biblioteke

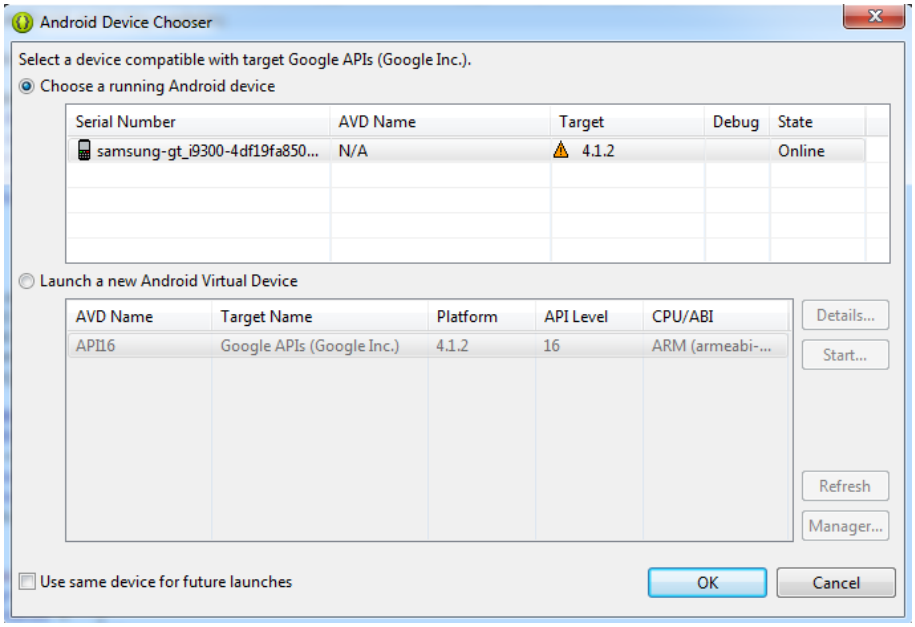
Biblioteke su grupirane u četiri grupe: Google APIs ili Android API ovisno o konfiguraciji projekta, Android ovisnosti (engl. d*ependencies*), referencirane biblioteke i sistemske biblioteke. Google APIs nudi mnoge dodatne funkcionalnosti u odnosu na Android API uključujući i OpenGL ES biblioteku. Referencirane biblioteke su biblioteke koje je programer dodao zbog svojih potreba.

Nakon uspješnog pakiranja (engl. *build*) aplikacije, kreira se *bin* direktorij koji sadrži prevedene Java klase, *Dalvik executable* datoteke, kompresiranu datoteku sa resursima i *apk* datoteku.

Aplikacija se može pokrenuti na Android emulatoru ili na pravom uređaju. Android emulator je virtualni uređaj koji omogućuje testiranje aplikacija bez korištenja pravih uređaja. [developer.android.com, 2013d]

Emulator je moguće dodati koristeći *Android Virtual Device Manager*. Svakom emulatoru je moguće prilagoditi hardverska svojstva poput radne memorije, veličine ekrana, interne memorije i dr. Osim hardverskih svojstva moguće je i odabrati ciljanu Android SDK verziju.

Jedan od mnogih načina da se Android aplikacija pokrene u *Eclipse* okruženju je desni klik na projekt te opcija *Run As -> Android application.* Odabir te opcije rezultira otvaranjem prozora kao na slici 18.



Slika 18. Pokretanje Android aplikacije

Otvoreni prozor nudi mogućnost odabira uređaja na kojem će se aplikacija instalirati. Na slici 18. je moguće odabrati virtualni uređaj pod nazivom „API16“ ili uređaj pod nazivom „samsung-gt..“. Da bi uređaj bio dostupan na ovoj listi, potrebno je konfigurirati sam uređaj te ga postaviti u način rada za razvoj što je moguće kroz opcije u *Settings -> Developer options.*

3.3. Android Activity

Android aplikacije se razvijaju na temelju četiri osnovna tipa komponenti : aktivnost (engl. *activity*), servis (engl. *service*), pružatelj sadržaja (engl. *content provider*) i primatelj sistemskih obavijesti (engl. *broadcast receiver*). [developer.android.com, 2013b]

Servis je komponenta koja služi za izvršavanje dugotrajnih procesa koji ne bi trebali blokirati glavnu dretvu, odnosno onemogućiti daljnju interakciju klijenta s aplikacijom. Servis ne pruža korisničko sučelje te se koristi na način da se proširi klasa *Service.*

Pružatelj sadržaja služi kao sučelje između različitih aplikacija i zajedničkih podataka. Ujedno postoje definirana prava prema kojima su definirane sigurnosne postavke. Postoji nekoliko implementiranih pružatelja sadržaja poput pružatelja za kontakte i kalendar. Iz različitih aplikacija, u slučaju da te aplikacije imaju potrebna prava, može se pristupiti kontaktima preko pružatelja sadržaja za kontakte. Pružatelji sadržaja se implementiraju na način da se proširi klasa *ContentProvider.*

Primatelj sistemskih obavijesti je komponenta koja najčešće služi za pokretanje servisa zbog određenog događaja. Događaji su reprezentirani kao obavijesti koje su emitirane na razini sustava. Mogu biti sistemske, poput „baterija je prazna“, ali mogu biti i emitirane iz aplikacija. Primatelji sistemskih obavijesti ne prikazuju korisničko sučelje.

Aktivnost je jedina komponenta koja omogućuje prikaz korisničkih sučelja, a time i interaktivnost korisnika s aplikacijom. Android aplikacija se najčešće sastoji od više aktivnosti od kojih je jedna definirana kao glavna, odnosno ona aktivnost koja se pokreće sa startanjem aplikacije. Svaka aktivnost može pokrenuti drugu aktivnost, ali se pritom trenutna aktivnost pauzira i stavlja u stog. Povratkom (*back button*) s nove aktivnosti na onu u stogu, nova se uništi i aktivnost sa stoga nastavi s radom. Aktivnost se kreira proširivanjem klase *Activity.*

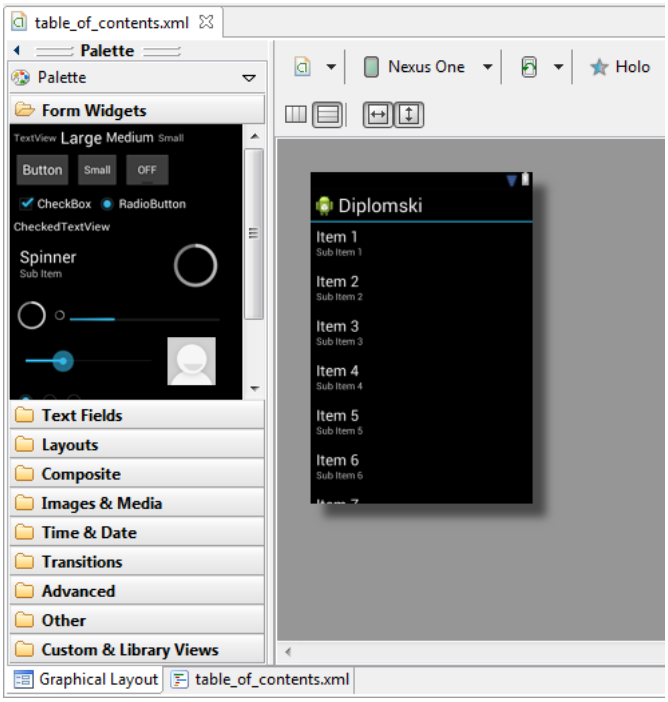
Aktivnost i klasu, koja implementira aktivnost, potrebno je navesti u *AndroidManifest.xml-u*. Na primjeru ispod je prikazana aktivnost *TableOfContents* i *OpenGLActivity* gdje je *TableOfContents* postavljena kao glavna aktivnost, odnosno ona koja se prikaže kada se aplikacija pokrene.

```
<activity android:name=".activity.TableOfContents"
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
<activity android:name=".activity.OpenGLActivity" android:label="@string/labelOpenGl" />
```

Kreiranje aktivnosti je odvojeno u dva koraka: kreiranje logike aplikacije i kreiranje korisničkog sučelja. Korisničko sučelje se kreira kroz XML konfiguraciju u *layout* direktoriju, kao u primjeru ispod.

```
<LinearLayout  mlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
    <ListView
         android:id="@android:id/list"
</LinearLayout>
```

XML konfiguracije je moguće urediti i u grafičkom prikazu kao na slici 19. Osnovni elementi za rad sa sučeljem su objekti koji proširuju *View*. *View* predstavlja pravokutni prostor na ekranu, koji je odgovoran za iscrtavanje i rad s događajima. Drugi bitan element su objekti koji proširuju klasu Layout te služe kao kontejner za *View* objekte.



Slika 19. Grafičko sučelje za kreiranje Android aktivnosti

Logika aplikacije se implementira u samoj implementaciji aktivnosti. Osim logike aplikacije, potrebno je povezati kreirano korisničko sučelje s aktivnosti, što je moguće u trenutku kada se aktivnost kreira. Povezivanje korisničkog sučelja s aktivnosti je moguće na sljedeći način.

```
setContentView(R.layout.table_of_contents);
```

U okviru u ispod je prikazano kako postaviti slušač (engl. *listener*) na određeni događaj. U konkretnom primjeru je postavljen slušač koji očekuje događaj „klik na element liste“ te kreira novu aktivnost *OpenGLActivity*.

```
setOnItemClickListener(new OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        final Intent launchIntent = new Intent(TableOfContents.this, OpenGLActivity.class);
        startActivity(launchIntent);
    }
});
```

Na okviru iznad je prikazan način kako kreirati i pokrenuti novu aktivnost.

4. Programsko rješenje

Kao primjer implementacije 3D grafike na mobilnim platformama izabran je OpenGL ES 2.0 na Android platformi. Glavna smjernica za izradu programskog rješenja je bila izrada aplikacije koja omogućuje prikaz 3D modela glavnog trga u Varaždinu. Odabrane tehnologije izabrane su iz razloga što je OpenGL standard za razvoj aplikacija s naprednim grafičkim elementima na mobilnim platformama. Specifično verzija 2.0 je izabrana jer je u vrijeme odabira teme diplomskog rada bila je najnovija i najzahtjevnija tehnologija na tom području. Kao takva dovoljno je motivirajuća za proučavanje i izrade programskog rješenja. Android platforma je odabrana iz razloga što je to otvoreni standard, te je autor rada imao pristup Android uređaju koji je realizaciju učinio mogućom.

Završna verzija programskog rješenja se temelji na klijent-server pristupu. Dakle, postoji serverska aplikacija s kojom klijentska aplikacija može komunicirati. Serverska aplikacija služi samo za posluživanje određenih konfiguracija koji definiraju izgled scene. Više o tome u poglavlju 4.2. Klijentska strana se sastoji od dvije aplikacije. Prva aplikacija služi isključivo za komuniciranje sa serverskom aplikacijom (nadalje klijentska aplikacija), dok druga Android aplikacija (nadalje Android aplikacija), koja konzumira klijentsku aplikaciju, služi za pripremu podataka i objekata potrebnih da bi se, koristeći OpenGL, iscrtao 3D model. Više o klijentskoj aplikaciji u poglavlju 4.3. te o Android aplikaciji u poglavlju 4.1.

4.1. Android aplikacija

Serverska i klijentska aplikaciju su samo pomoćne aplikacije koje poslužuju podatke za Android aplikaciju. Android aplikacija je srž ovog rada iz razloga što je u njoj prikazan način kako se OpenGL ES 2.0 može koristiti u mobilnim aplikacijama. Android aplikacija je mobilna aplikacija koja je kompatibilna s mobilnim uređajima koji rade na Android platformi. S obzirom da se koristi novija inačica OpenGL ES, samo određeni uređaji imaju i hardversku podršku kako za OpenGL tako i za ovu aplikaciju. U ovom poglavlju će biti detaljno objašnjen rad s OpenGL ES-om, te kako je implementirana logika koja konzumira XML konfiguraciju i na temelju nje kreira sučelje s 3D modelom. Također, biti će objašnjen rad s OrmSqlLite te ostale metode i tehnike koje su korištene za izradu ove aplikacije.

4.1.1. Implementacija

4.1.1.1.OpenGL ES 2.0 i Android

Android platforma sadrži klasu *Activity* koja omogućuje kreiranje raznih prilagođenih sučelja te se brine o svim interakcijama s krajnjim korisnikom. Također, moguće je pomoću metode *setContentView(View view)* eksplicitno postaviti sam *view*, odnosno osnovni kontejner za komponente korisničkog sučelja. Pomoću te metode se u određeni *Activity* postavlja *view* koji nasljeđuje OpenGL karakteristike.

GLSurfaceView je klasa koja indirektno proširuje *View* klasu te se kao takva može postaviti kao *contentView* za neki *Activity*. Ali to nije sve, *GLSurfaceView* pripremi posebno sučelje i konfigurira EGL sučelje čime omogućuje OpenGL-u da iscrtava na to sučelje. Međutim, kroz *GLSurfaceView* se ne izvodi nikakvo iscrtavanje. Za to je predviđeno sučelje *Renderer*, koje se postavlja pomoću metode *setRenderer(Rendere renderer)* .

Na primjeru se može vidjeti klasa koja proširuje *Activity* i postavlja drugi *View*. Posebno su zanimljive sljedeće tri metode:

- *onCreate* metoda se poziva kada se aktivnost starta;
- *onPause* metoda se poziva kada aktivnost ide u pozadinu, odnosno nije dostupna klijentu, ali nije izbrisana iz memorije;
- *onResume* metoda se najčešće poziva nakon metode *onPause*, kada je aktivnost ponovo dostupna klijentu.

Navedene tri metode su proširene s funkcionalnošću da izvršavaju metode istih naziva na objektu *OpenGLSurfaceView*.

```
public class OpenGLActivity extends Activity {
        private OpenGLSurfaceView mGLView;
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                // instantiate and setContentView
                mGLView = new OpenGLSurfaceView(this, city);
                setContentView(mGLView);
        }
        @Override
```

```
        protected void onPause() {
                super.onPause();
                mGLView.onPause();
        }
        @Override
        protected void onResume() {
                super.onResume();
                mGLView.onResume();
        }
}
```

U sljedećem okviru prikazan je *OpenGLSurfaceView* klasa koja proširuje *GLSurfaceView*. *GLSurfaceView*, osim što OpenGL-u pruža način za iscrtavanje, omogućuje se upravljanje događajima na korisničkom sučelju. O tome detaljnije u poglavlju 4.1.1.4. Bitan detalj u ovom primjeru je postavljanje *Renderer* objekta, jer je on zadužen za iscrtavanje na korisničko sučelje.

```
public class OpenGLSurfaceView extends GLSurfaceView {
        public OpenGLSurfaceView(Context context, City city) {
                super(context);
                // Create an OpenGL ES 2.0 context.
                setEGLContextClientVersion(2);
                // Set the Renderer for drawing on the GLSurfaceView
                mRenderer = new OpenGLRenderer();
                mRenderer.setContext(context);
                setRenderer(mRenderer);
                // Render the view only when there is a change in the drawing data
                setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
        }
```

OpenGLRenderer je klasa koja proširuje *Renderer* klasu i ona iscrtava sadržaj na sučelje. Posebno su zanimljive sljedeće metode:

- *onSurfaceCreated* se poziva kada se sučelje kreira ili ponovno kreira, odnosno svaki put kada se EGL kontekst izgubi, te je potrebno ponovo zauzeti resurse;

- *onDrawFrame* je odgovorna za iscrtavanje trenutne slike (engl. *frame*);
- *onSurfaceChanged* se poziva kada se sučelje kreira ili kada se njegova veličina promijeni.

Primjer pokazuje ono što je potrebno implementirati, ali ne i kako. Detaljnije o tome u poglavlju 4.1.1.10.

```
public class OpenGLRenderer extends GLSurfaceView.Renderer {
        @Override
        public void onSurfaceCreated(GL10 unused, EGLConfig config) {
                // Sets background color to black
                GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        // More code that prepare buffers, allocate memory, load textures, etc..
        }
        @Override
        public void onDrawFrame(GL10 unused) {
                // Clears color buffer
                GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);
        // More code that calculate projectionMatrix, camera and draw objects..
        }
        @Override
        public void onSurfaceChanged(GL10 unused, int width, int height) {
                // Defines projection matrix in terms of six clip planes
                GLES20.glViewport(0, 0, width, height);
                Matrix.frustumM(mProjMatrix, 0, left, right, bottom, top, near, far);
        }
}
```

4.1.1.2.Matrice

U prethodnom poglavlju je objašnjeno kako omogućiti da OpenGL iscrta model na Android sučelju, ali je izostavljena implementacija. Kako bi to sve skupa funkcioniralo mora se definirati projekcijska matrica i matrica kamere.

Projekcijska matrica

Projekcijska matrica se definira na sljedeći način:

```
private final float [] mProjMatrix = new float[16];
@Override
public void onSurfaceChanged(GL10 unused, int width, int height) {
```

```
        GLES20.glViewport(0, 0, width, height);
        final float ratio = (float) width / height;
        final float left = -ratio;
        final float right = ratio;
        final float bottom = -1.0f;
        final float top = 1.0f;
        final float near = 1.0f;
        final float far = 1000.0f;
        Matrix.frustumM(mProjMatrix, 0, left, right, bottom, top, near, far);
}
```

OnSurfaceChanged metoda se nalazi u klasi *Renderer*. S obzirom da se projekcijska matrica brine o tome da sučelje bude proporcionalno dimenzijama ekrana, ovo je idealno mjesto za kreiranje projekcijske matrice. Kada se promijeni veličina sučelja, postavljaju se nove vrijednosti za projekcijsku matricu. Projekcijskom matricom definira se prostor koji se iscrta na sučelju. Sve što je izvan ovih ograničenja, neće se iscrtati. Npr. definirali smo udaljenu odrezujuću plohu (engl. *far clipping plane)* s vrijednošću tisuću. To znači da će svi objekti koji su na udaljenosti većoj od tisuću,od trenutnog položaja kamere, biti odbačeni i neće se uzeti u obzir za prikaz na ekranu. Projekcijska matrica kreira se pomoću metode *Matrix.frustrumM().*

Matrica kamere

Matrica kamere sadrži vektore koji opisuju položaj kamere, smjer gledanja i *up-vector*. Kreira se pomoću metode *Matrix.setLookAtM().* Matricu kamere je potrebno kreirati svaki put kod iscrtavanja nove slike. U sljedećem poglavlju je detaljno opisano kako se koristi matrica kamere za simuliranje kretanja, kao posljedica korisničke interakcije. Druga linija u *onDrawFrame* metodi množi matricu kamere s projekcijskom matricom i rezultat množenja se sprema u *mMVPMatrix* objekt.

```
private final float [] cameraMatrix = new float [16];
@Override
public void onDrawFrame(GL10 unused) {
        Matrix.setLookAtM(cameraMatrix, 0, px, py, pz, ex, ey, ez, 0, 0, 1);
        Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, cameraMatrix, 0);
}
```

Matrica *mMVPMatrix* bit će kasnije proslijeđena u program za sjenčanje vrhova te će procesor vrhova izvršiti transformacije i izračunati konačne koordinate objekata.

4.1.1.3.Sustav kamere

Prethodno je objašnjeno zašto se koristi matrica kamere, ali nije objašnjena logika. U nastavku je objašnjeno kako je implementirano popunjavanje matrice s odgovarajućim podacima.

U aplikaciji se položaj kamere i smjer gledanja može mijenjati kroz javne varijable (*px, py, pz* i *phi*). Varijable *px, py,* i *pz* su koordinate položaja kamere te ujedno čine vektor položaja kamere. Varijabla *phi* označava kut u radijanima, odnosno orijentaciju kamere u *xy* ravnini i služi za izračun točke u koju je kamera usmjerena. Vektor, koji reprezentira smjer gledanja kamere, se kreira od privatnih varijabli *ex, ey, ez*. S obzirom da su varijable privatne ne može se direktno utjecati na smjer gledanja kamere, već ga je moguće korigirati samo preko varijable *phi*. Uz pomoć jednostavne trigonometrije izračunavaju se koordinate *ex* i *ey*, a koordinata *ez* ima uvijek vrijednost 2.5 (to je visina na kojoj se nalazi kamera).

```
public float px = 40.0f, py = 40.0f, pz = 2.5f, phi = 0.0f;
@Override
public void onDrawFrame(GL10 unused) {
        float d = 100.0f;
        float ex = px + (float) (d * Math.sin(phi));
        float ey = py + (float) (d * Math.cos(phi));
        float ez = 2.5f;
        Matrix.setLookAtM(cameraMatrix, 0, px, py, pz, ex, ey, ez, 0, 0, 1);
        Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, cameraMatrix, 0);
}
```

Korisniku je omogućeno da se kreće i mijenja smjer pogleda samo u koordinatnom sustavu xy. S obzirom da je za *view-up* vektor postavljen (0, 0, 1), korisniku će izgledati kao da se kreće u horizontalnoj ravnini i može se okretati lijevo – desno.

4.1.1.4.Android *Touch eventi* i simuliranje kretanja

U prijašnjim poglavljima je opisana svrha *GLSurfaceView* klase kao mosta između OpenGL-a i Android sučelja. Osim toga, služi i za hvatanje korisničkih akcija. To je omogućeno kroz

metodu *boolean onTouchEvent(MotionEvent event). MotionEvent* je objekt koji nosi informacije o vrsti korisničke akcije. U ovom kontekstu se odnosi na akcije prstom na korisničkom sučelju. Također nosi informacije o tipu akcije prstom te koordinatama na kojima se događaj dogodio.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
        if (event == null || mRenderer == null)
                return true;
        if (event.getAction() == MotionEvent.ACTION_MOVE) {
                if (event.getPointerCount() == 1) {
                        onRotateEvent(event);
                } else if (event.getPointerCount() == 2) {
                        onMoveEvent(event);
                }
        }
        if (event.getPointerCount() == 1 || event.getPointerCount() == 2) {
                mPreviousX = event.getX();
                mPreviousY = event.getY();
        }
        requestRender();
        return true;
}
```

Na okviru iznad je prikazano kako iz događaja razlikovati akcije. U ovoj implementaciji je ideja da korisnik ima dvije akcije:

- Kretanje – povlačenjem dva prsta po ekranu;
- Rotiranje - povlačenjem jednog prsta po ekranu.

U primjeru metode *onTouchEvent* prvo se provjeri da li je događaj valjan. Točnije, provjeri se da li referenca postoji (*event != null*) i da li je događaj tip *MotionEvent.ACTION_MOVE*. Tip *MotionEvent.ACTION_MOVE* označava kretnju koja ima početnu i završnu koordinatu, koje reprezentiraju početak i kraj pokreta po ekranu. Objekt tipa *MotionEvent* nosi podatak o tome koliko je prstiju bilo na ekranu prilikom izvođenja te akcije. Taj podatak se dobije kao rezultat

metode *event.getPointerCount()*, te se na temelju tog podatka odlučuje da li je riječ o kretanju ili rotiranju. Nakon što se odradi manipulacija ovisno o tipu događaja, postavljaju se *mPreviousX* i *mPreviousY* koordinate na trenutne koordinate, te se poziva *requestRender()* metoda koja zatraži kreiranje nove slike.

Gore prikazani primjer poziva različite metode, ovisno o tipu akcije. U prethodnom poglavlju je objašnjeno kako kamera može pozicionirati te kako se mijenja njen smjer gledanja. Na taj način se simulira kretanje po 3D modelu. Kretanje rezultira pomicanjem kamere po 3D modelu, dok rotiranje rezultira promjenom smjera gledanja kamere. Kretanje se realizira promjenom vektora koji opisuje položaj kamere, a rotiranje promjenom vektora koji opisuje kuda kamere gleda.

U sljedećem okviru su prikazane implementacije metoda za kretanje i rotiranje.

```
private static int CAMERA_MOVE_FACTOR = 25;
private static int CAMERA_ANGLE_ROTATION_FACTOR = 1;
private void onRotateEvent(MotionEvent event) {
        float deltaX = (event.getX() - mPreviousX) / this.getWidth();
        mRenderer.phi += deltaX * CAMERA_ANGLE_ROTATION_FACTOR;
}
private void onMoveEvent(MotionEvent event) {
        float deltaY = (event.getY() - mPreviousY) / this.getHeight();
        double px = deltaY * CAMERA_MOVE_FACTOR * Math.sin(mRenderer.phi);
        double py = deltaY * CAMERA_MOVE_FACTOR * Math.cos(mRenderer.phi);
if (!mRenderer.getShapeContainer().isCollisionDetected(new PointXY(mRenderer.px + px,
mRenderer.py + py))) {
                mRenderer.px += px;
                mRenderer.py += py;
        }
}
```

CAMERA_MOVE_FACTOR i *CAMERA_ANGLE_ROTATION_FACTOR* su statičke varijable. One predstavljaju faktore koji definiraju koliki će biti omjer razlike u koordinatama na ekranu i razlike u otklonu kamere. Te vrijednosti se u aplikaciji mogu konfigurirati jer na

različitim uređajima se različito upravlja s događajima na ekranu. Moguće je da na određenim uređajima kretanje bude prebrzo ili presporo.

onRotateEvent

Metoda *onRotateEvent* računa koliki je zaokret kamere potrebno napraviti. U obzir uzima širinu ekrana, odnos trenutne koordinate prsta na ekranu te prethodno zapamćene koordinate. Korisnik može okretati kameru samo u smjeru lijevo – desno. Shodno tome jedino pomak prstom lijevo - desno služi za okretanje kamere, a pomak gore – dolje se zanemaruje. Ta vrijednost korigira se faktorom *CAMERA_ANGLE_ROTATION_FACTOR* i ažurira se vrijednost varijable *phi* u *Renderer* klasi. Metoda *requestRender()* će zatražiti prikaz nove slike, što rezultira pozivom metode *onDrawFrame()*. S obzirom da se vrijednost *phi* varijable promijenila, matrica kamere će imati nove vrijednosti, što će utjecati na izgled slike.

onMoveEvent

Metoda onMoveEvent računa koliki je pomak kamere potrebno napraviti. U obzir se uzima visina ekrana, razlika između zapamćene vrijednosti iz prethodnog događaja te trenutne vrijednosti. U metodi *onRotateEvent* jedino je pomak po lijevo – desno bio zanimljiv, dok u ovoj metodi jedino pomak prstima gore –dolje ima svrhu. Pomaci lijevo – desno se zanemaruju jer je korisniku omogućeno kretanje samo naprijed – natrag u smjeru gledanja kamere. Promjene koordinata se računaju po sljedećim formulama,

$$\text{novaKoordinataX} = \text{deltaY} * CAMERA\_MOVE\_FACTOR * sin(\text{phi})$$

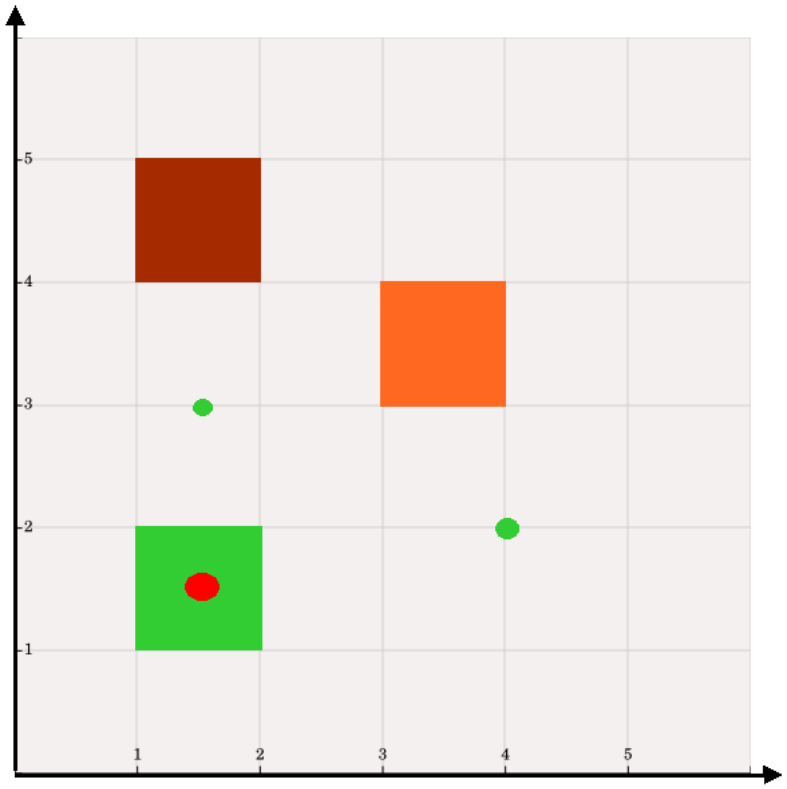
$$\text{novaKoordinataY} = \text{deltaY} * CAMERA\_MOVE\_FACTOR * cos(\text{phi})$$

a nove koordinate mogu dobiti pribrajanjem na trenutnu poziciju kamere. Prije nego se postave nove koordinate kamere u klasu *Render*, poziva se metoda *isCollisionDetected* na svakom objektu u sceni. Detaljnije o detekciji sudara u poglavlju 4.1.1.5. Ako se sudar detektira, tada se novoizračunate koordinate kamere zanemaruju.

4.1.1.5.Detekcija sudara

Detekcija sudara je veoma opširno i kompleksno područje ako je riječ o 3D detekciji sudara. Činjenica da je korisnik ograničen na kretanje samo u horizontalnoj ravnini pojednostavljuje problem detekcije sudara. Unatoč tome što je model trodimenzionalan, kada je u pitanju detekcija sudara, on se svodi na dvodimenzionalan sustav.

Detekcija sudara se koristi kako bi se spriječilo kretanje kroz objekte. Korisnik se kreće tako da se ovisno o njegovim akcijama na ekranu računaju nove koordinate kamere. Prilikom svakog izračuna potrebno je provjeriti da li se nova koordinata nalazi unutar nekog objekta. Za svaki tip objekta (*cube, pyramid..)* je definirana bazna stranica, odnosno objekt određen vrhovima koji leže u horizontalnoj ravnini. Taj objekt predstavlja područje u koje kamera ne smije ući, odnosno novo izračunate koordinate ne smiju biti unutar baznih stranica objekata u modelu. Ako se novo izračunata koordinata nalazi unutar takvog područja, detektira se sudar i takva koordinata nije valjana, te se ona odbacuje.



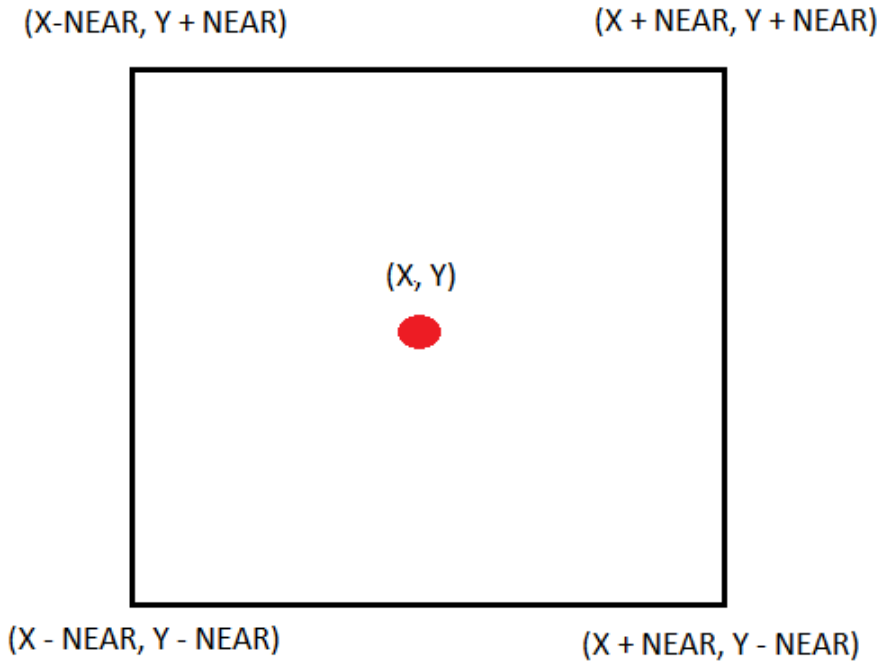
Slika 20. Detekcija sudara

Na slici 20. je primjer jednostavnog modela gdje obojani kvadrati predstavljaju bazne stranice objekata u koordinatnom sustavu xy. Obojani kružići predstavljaju novo izračunate koordinate kamere. U ovom slučaju će se sudar detektirati ako je novo izračunata koordinata unutar baznih stranica objekata. Evo nekoliko primjera novo izračunatih koordinata kamere:

- (4, 2) – valjana koordinata;
- (1.5, 1.5) – koordinata se nalazi unutar zelenog kvadrata, odnosno unutar bazne stranice tog objekta te nije valjana;
- (1.5, 3) – valjana koordinata.

Za izračun detekcije sudara koristi se jednostavan algoritam koji provjerava da li se točka nalazi unutar poligona koji je definiran x i y koordinatama. Metoda *CollisionDetectorImpl. isPointInPolygon* implementira taj algoritam. [stackoverflow.com 2013]

OpenGL koristi projekcijsku matricu kojom je definirano šest ploha. Plohe definiranju područje koje će se prikazati. Jedna od tih šest ploha je bliža odrezujuća ploha (engl. *near clipping plane*) koja ograničava prikaz objekata preblizu kameri. Npr. ako se kamera postavi neposredno blizu objektu, koji predstavlja zgradu, bliža odrezujuća ploha bi onemogućila iscrtavanje tog objekta ili dijela objekta. Takvo ponašanje asocira korisnika da je ušao u zid zgrade. Iz tog razloga, kretanje po modelu je dodatno ograničeno. Problem je riješen na način da se ne radi detekcija sudara pomoću točne koordinate kamere, već da se lokacija kamere reprezentira kao poligon koji ima neku površinu. Zbog jednostavnosti implementacije, nije implementiran dodatni algoritam za provjeru preklapanja dva poligona, već se postojećim algoritmom za provjeru točke unutar poligona, provjeri svaki vrh poligona koji reprezentira položaj kamere. Poligon, koji reprezentira kameru, je kreiran tako da se pomoću koordinata kamere izračunaju vrhovi na udaljenosti većoj od definirane vrijednosti bliže odrezujuće plohe. Na slici 21. je prikazano kako su izračunati vrhovi poligona kamere.



Slika 21. Poligon kamere

*4.1.1.6.*Rad sa *Sql OrmLite*

Object Relational Mapping Lite (ORM Lite) omogućava neke jednostavne funkcionalnosti za trajno spremanje Java objekata u SQL bazu podataka, izbjegavajući kompleksne i zahtjevne funkcionalnosti standardnih ORM paketa. [ormlite.com, 2013] Za rad s *ORMLite* potrebno je nekoliko koraka:

- Preuzeti dvije datoteke *ormlite-android* i *ormlite-core;*
- Postaviti ih u putanju projekta;
- Kreirati modele;
- Za svaki kreirani model kreirati *DAO (Data Access Object)* i referencirati u implementaciji *OrmLiteSqliteOpenHelper;*
- Implementirati *OrmLiteSqliteOpenHelper*.

Kreiranje modela se svodi na kreiranje *JavaBean-ova*, odnosno klasa koje implementiraju sučelje *Serializable*, imaju privatne atribute, konstruktor bez argumenata, te *gettere* i *settere* za atribute. Jedina razlika je što je s *OrmLite* anotacijama potrebno označiti klasu i atribute. Anotacija *@DatabaseTable* konfigurira da klasa *City* bude sačuvana u bazi kao tablica *city*, anotacija *@DatabaseField* preslikava atribut *id* u kolonu istog naziva.

```
@DatabaseTable
public class City implements Serializable{
        private static final long serialVersionUID = 1L;
        @DatabaseField(id = true)
        private String id;
        public City(){}
        //geters and setters
```

Kreiranje *DAO-a* se svodi na pozivanje metode *getDao()*, ali za kreiranje *DAO-o* s posebnim metodama potrebno je kreirati klasu koja proširuje *BaseDaoImpl*. Prilikom proširivanja klase potrebna su dva parametra, klasa s kojim je preslikana tablica i klasa ID kolone.

```
public class CityDataService extends BaseDaoImpl<City, Integer>{
        protected CityDataService(ConnectionSource connectionSource) throws
SQLException {
```

```
            super(connectionSource, City.class);
        }
}
```

Klasa *DatabaseHelper* služi da kreira bazu prema programerovim instrukcijama. Naziv i verzija baze se mogu konfigurirati. Prilikom pokretanja aplikacije, ako ne postoji baza tog naziva i verzije, se izvrši metoda *oncreate()*. U okviru niže *oncreate()* metoda kreira tablicu prema anotacijama u *City.class*. Osim toga pogodno je ovdje navesti sve *DAO* klase. U sljedećem okviru je prikazano kako se omogućuje rad s *DAO* objektima.

```
getDao().getCityDao().findAll().
```

```
public class DatabaseHelper extends OrmLiteSqliteOpenHelper {
        private static final String DATABASE_NAME = "dtopler.db";
        private static final int DATABASE_VERSION = 27;
        private Dao<City, String> cityDao;
        public DatabaseHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }
        @Override
        public void onCreate(SQLiteDatabase sqLiteDatabase, ConnectionSource
connectionSource) {
            TableUtils.createTable(connectionSource, City.class);
        }
        @Override
        public void onUpgrade(SQLiteDatabase sqLiteDatabase, ConnectionSource
connectionSource, int oldVersion, int newVersion) {
            TableUtils.dropTable(connectionSource, City.class, true);
            onCreate(sqLiteDatabase, connectionSource);
        }
        public Dao<City, String> getCityDao() throws SQLException {
            return getDao(City.class);
        }
}
```

4.1.1.7.Spremanje XML konfiguracije

Korisniku Android aplikacije omogućeno je da ažurira svoje lokalne podatke s podacima koji su izloženi na servisu kao *zip* arhiva. U poglavlju 4.2.1.4 je objašnjeno kakvu hijerarhiju te *zip* arhive moraju imati. Prvobitna ideja je bila da se takva *zip* arhiva preuzme te spremi u identičnoj hijerarhiji na datotečni sustav mobilnog uređaja. U tom slučaju bi se XML konfiguracija uvijek čitala i direktno iz te konfiguracije kreirali objekti koji bi se potom iscrtali. Nakon malo eksperimentiranja uočeno je da je parsiranje veoma procesorski zahtjevno, te samim time vremenski dugo na mobilnim uređajima. To bi značilo da je svaki put kada klijent želi vidjeti 3D model, potrebno parsirati XML datoteku, kreirati objekte i prikazati model. Druga opcija je korištenje My*SQL* baze. Uočeno je da spremanje u bazu traje značajno duže nego spremanje XML konfiguracije, ali čitanje iz baze je i do tri puta brže nego čitanje XML datoteke. Iz tog razloga postoje dvije implementacije: prva koja koristi XML konfiguraciju, a druga koja prilikom preuzimanja *XML* datoteke istu pročita te spremi cijelu konfiguraciju u bazu. U završnoj verziji Android aplikacije se koristi verzija koja radi sa SQL bazom.

UpdateActivity je aktivnost koja je zadužena za sve gore navedeno. Za komunikaciju sa servisom i preuzimanje *zip* datoteke koristi se *Client* objekt iz klijentske aplikacije. Metoda *FileClient.getStreamFromPath* s parametrom *resourcesUrl* vraća objekt tipa *DownloadDescriptor* koji ima opis konekcije, odnosno *InputStream.* Pomoću *InputStream-a* kreiramo objekt *ZipInputStream* koji je specijalna verzija *InputStream-a* i služi za prijenos i raspakiravanje zip datoteka. *ZipInputStream* ima metodu *getNextEntry()* koja iz *stream*-a prepoznaje datoteke i direktorije. Time se direktno iz *InputStream-a* mogu u specifičan *OuptuStream* proslijediti podaci. Npr. pomoću *getName()* metode se dobije izvorna putanja u zip arhivi. Na temelju toga se odredi putanja na uređaju na koju želimo spremiti tu datoteku. Uz pomoć *FileOutputStream(String path)* se kreira *OutputStream* koji sprema proslijeđene podatke u datoteku.

```
Client client = new ClientFactory().getClient();
DownloadDescriptor dd = client.getFileClient().getStreamFromPath(city.getResourcesUrl());
ZipInputStream zis = new ZipInputStream(new BufferedInputStream(dd.getInputStream()));
```

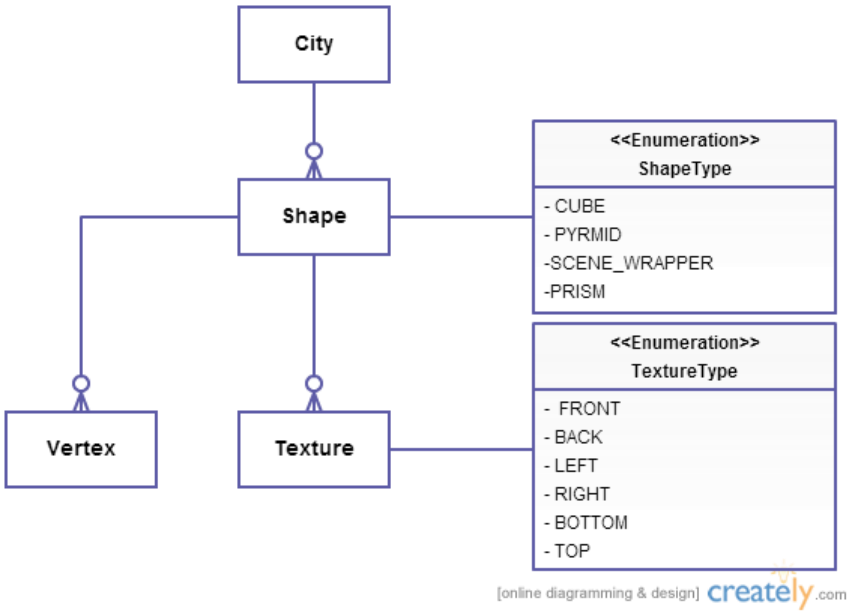
U okviru iznad se vidi da je *Client* objekt kreiran kroz *ClientFactory*. *ClientFactory* se koristi kako bi na jednom mjestu u aplikaciji bilo zbrinuto kreiranje tog klijenta. U okviru ispod je

prikazano kako ta klasa izgleda (u toj klasi je eksplicitno zadana putanja do servisa što obično nije dobra praksa).

```
public class ClientFactory {
	private static final String url = "http://diplomski-dtopler.rhcloud.com";
	private Client client;
	public Client getClient(){
		ClientConfig clientConfig = new ClientConfig();
		clientConfig.setUrl(url);
		client = new Client(clientConfig);
		return client;
	}
}
```

Nakon uspješnog preuzimanja i spremanja *zip* arhive, potrebno je obrisati postojeće podatke za taj grad, pročitati XML konfiguraciju te spremiti nove podatke u bazu.

Na slici 22. su prikazani svi objekti koje treba kreirati i spremiti u bazu prema opisu u XML datoteci. *City* objekt sadrži podatke relevantne za neki grad, kao i putanju od direktorija na serveru, gdje se nalazi *zip* konfiguracija za taj grad. Taj podatak se koristi prilikom ažuriranja lokalnog repozitorija. Objekt *Shape* se koristi za definiranje jednog objekta na modelu, tipično za zgradu, krov i sl. *Shape* objekt ima referencu na *City* objekt te sadrži općenite informacije o objektu koji će se iscrtavati. Prema tome, jedan grad, odnosno jedan model može sadržavati više *Shape* objekata. Također sadrži informaciju o tipu objekta. Tipovi su na slici prikazani kao *ShapeTypeEnumeration*. Objekt *Texture* opisuje teksturu. Nosi podatak o putanji na kojoj se nalazi slika te o tipu teksture koji je na slici prikazan u tablici *TextureTypeEnumearation*. Objekt *Shape* može imati više tekstura, ovisno o tipu objekta. Npr. *ShapeType.CUBE* može imati šest tekstura, dok *Shape.Type.PYRAMID* može imati pet tekstura. Objekt *Vertex* sadrži podatke o točkama koje predstavljaju vrhove objekta u obliku x, y i z koordinate. Objekt *Shape* može imati više *Vertex-a*, opet zavisno o tipu. *Enumeration* objekti nisu spremljeni u bazu, već su implementirani kao *Enum* klase.



Slika 22. Era model

Brisanje svih podataka je implementirano u *ShapeDataService.deleteAllByCityId(String cityId)* metodi. Logika je da se pronađu svi *Shape* objekti koji imaju referencu na *City* objekt prema parametru te da se za svaki od njih pronađu ostali objekti (*Vertex* i *Texture*) na isti način.

Spremanje podataka opisano je u sljedećim koracima:

- **Parsirati datoteku**

```
Document xml = XmlUtils.parseXML(is);
```

- **Svaki *Shape* objekat obraditi posebno**

```
List<String> subnodesPaths = XmlUtils.getSubnodesPaths(xml,
"modelDescription/shapes");
for (String root : subnodesPaths) {
        Shape shape = new Shape();
        shape.setCityId(city.getId());
        shape.setName(name);
        shape.setType(type.getId());
        getHelper().getShapeDataService().create(shape);
        saveVertices(xml, root, shape);
        saveTextures(xml, root, shape);
}
```

- **Pronaći sve vrhove u *shape-u* i spremiti u bazu – metoda *saveVertices***

```
List<String> verticesPath = XmlUtils.getSubnodesPaths(xml, root + "/vertices");
for (String path : verticesPath) {
        Vertex vertex = new Vertex();
        float x = Float.parseFloat( XmlUtils.getNode(xml, path).getAttributes().
getNamedItem("x").getNodeValue());
        vertex.setX(x);
        //isto za ostale koordinate
        getHelper().getVertexDataService().create(vertex);
}
```

- **Pronaći sve teksture i spremiti ih u bazu – metoda *saveTextures***

```
List<String> texturesPath = XmlUtils.getSubnodesPaths(xml, root + "/textures");
for (String xmlPath : texturesPath) {
        TextureType type = TextureType.findType(XmlUtils.getNode(xml,
xmlPath).getAttributes().getNamedItem("location").getNodeValue());
        Node nodeRepeat = XmlUtils.getNode(xml,
xmlPath).getAttributes().getNamedItem("repeatTexture");
        String path = XmlUtils.getNodeTextContent(xml, xmlPath + "/path");
        Texture texture = new Texture();
        texture.setRepeat( nodeRepeat == null ? 1 : I
        Integer.parseInt(nodeRepeat.getNodeValue()));
        texture.setType(type.getId());
        texture.setPath(path);
         texture.setShapeId(shape.getId());
        getHelper().getTextureDataService().create(texture);
        }
```

4.1.1.8.GLUtils

GLUtils je klasa sa statičkim metodama koje se koriste za učitavanje tekstura, programa i programa za sjenčanje[2]. *GLUtils* klasa ima svoju primjenu samo u klasi *Renderer* ali je neophodna za iscrtavanje bilo kakvog sadržaja koristeći *OpenGL*. Sljedeće metode su dio ove klase:

- *int createTexture(String path, Context context);*
- *int loadProgram(String strVSource, String strFSource);*
- *int loadShader(String strSource, int iType).*

Metoda *createTexture*

Metoda *createTexture* izgleda kao na sljedećem primjeru te se vidi da koristi *FileLoader* klasu. Pomoću iste se na temelju putanje dobije *Bitmap* opis teksture te pozove privatnu metodu *createTexture(Bitmap bitmap, Context context).*

```
public static int createTexture(String path, Context context){
        FileLoader fileLoader = new FileLoader();
        Bitmap bitmap = fileLoader.getBitmap(path);
        return createTexture(bitmap, context);
}
```

FileLoader klasa sadrži logiku za čitanje datoteke s datotečnog sustava samog uređaja. Privatna metoda *createTexture* koristi metode iz *GLES20* biblioteke za kreiranje teksture. Na primjeru su prikazani bitni detalji privatne *createTexture* metode.

```
final int[] textureHandle = new int[1];
return textureHandle[0];
```

U *OpenGL-u* prvo je potrebno kreirati kontejner koje će sadržavati podatke o samoj teksturi, odnosno samu sliku. Kontejner kreiramo na sljedeći način:

```
GLES20.glGenTextures(1, textureHandle, 0);
```

U *textureHandle* varijablu je generiran identifikator kontejnera. Nakon što je kontejner kreiran potrebno je *OpenGL-u* dati naredbu da se taj kontejner treba koristiti. Naredba za odabir kontejnera je sljedeća:

[2] Dio tih metoda je preuzet sa [Google Inc., 2013]

```
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureHandle[0]);
```

Željeni kontejner je odabran na temelju identifikatora kontejnera koji je kreiran prvim korakom, te kojeg proslijedimo u *glBindTexture* metodu kao drugi parametar. Sada *OpenGl* ima pod kontrolom odabrani kontejner, može se zaista učitati i sama slika kao na primjeru ispod.

```
GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0);
```

Metoda *createTexture* vraća upravo generirani identifikator kontejnera za kojeg će taj kontejner ostati vezan dok se kontejner ne obriše.

Metoda *loadProgram*

Metoda *loadProgram* služi za kreiranje programa, odnosno kreiranje kontejnera koji sadrži prevedene programe za sjenčanje. Metoda *loadProgram* prima dva parametra: prvi je program za sjenčanje vrhova, a drugi je program za sjenčanje točaka. Programe za sjenčanje je prvo potrebno učitati, to se izvrši pomoću metode *loadShader*, zatim se uz pomoć *GLES20* metoda izvrše sljedeći koraci.

- **Kreirati kontejner i kreirati identifikator**

```
iProgId = GLES20.glCreateProgram();
```

- **Dodati programe za sjenčanje u kontejner**

```
GLES20.glAttachShader(iProgId, iVShader);
GLES20.glAttachShader(iProgId, iFShader);
```

- **Povezati programe za sjenčanje**

```
GLES20.glLinkProgram(iProgId);
```

- **Provjeriti da li je povezivanje uspješno završeno**

```
GLES20.glGetProgramiv(iProgId, GLES20.GL_LINK_STATUS, link, 0);
```

- **Izbrisati programe za sjenčanje**

```
GLES20.glDeleteShader(iVShader);
```

```
GLES20.glDeleteShader(iFShader);
```

Metoda *loadProgram* vraća identifikator kontejnera, odnosno programa.

Metoda *loadShader*

Metoda *loadShader* prevodi program za sjenčanje koji je potrebno prevesti ovisno o tipu programa za sjenčanje (*GLES20.GL_VERTEX_SHADER* ili *GLES20.GL_FRAGMENT_SHADER*). Kako bi se program za sjenčanje preveo potrebno je napraviti sljedeće korake:

- **Kreirati program za sjenčanje**

```
int iShader = GLES20.glCreateShader(iType);
```

- **Dodati izvorni kod programa za sjenčanje**

```
GLES20.glShaderSource(iShader, strSource);
```

- **Prevesti program za sjenčanje**

```
GLES20.glCompileShader(iShader);
```

- **Provjeriti status prevođenja**

```
GLES20.glGetShaderiv(iShader, GLES20.GL_COMPILE_STATUS, compiled, 0);
if (compiled[0] == 0) {
        //error happen
  }
```

Metoda *loadShader* vraća identifikator generiranog programa za sjenčanje.

4.1.1.9.Programi za sjenčanje [3]

Već je spomenuto da postoje dvije vrste program za sjenčanje, program za sjenčanje vrhova i točaka. U ovom poglavlju će biti opisani programi za sjenčanje koji se koriste u aplikaciji.

Program za sjenčanje vrhova

```
attribute vec4 a_position;
attribute vec2 a_texCoords;
```

[3] Programi za sjenčanje koji se koriste u aplikaciju su preuzeti sa [Google Inc., 2013]

```
uniform mat4 u_VPMatrix;
varying vec2 v_texCoords;
void main()
        v_texCoords = a_texCoords;
        gl_Position = u_VPMatrix * a_position;
}
```

Ulazni parametri prikazanog program za sjenčanje su:

- *a_position* - koordinate vrhova objekta koji se iscrta;
- *a_texCoords* - koordinate tekstura;
- *u_VPMatrix* - matrica transformacije (umnožak projekcijske matrice i matrice kamere).

U *main* metodi se izvrše dvije akcije:

- Ulazni atribut *a_texCoords* se upiše u atribut *v_tecCoords* koji se prosljeđuje u program za sjenčanje točaka.*;*
- Izračuna se atribut *gl_Position* koji predstavlja konačne koordinate vrha.

Program za sjenčanje točaka

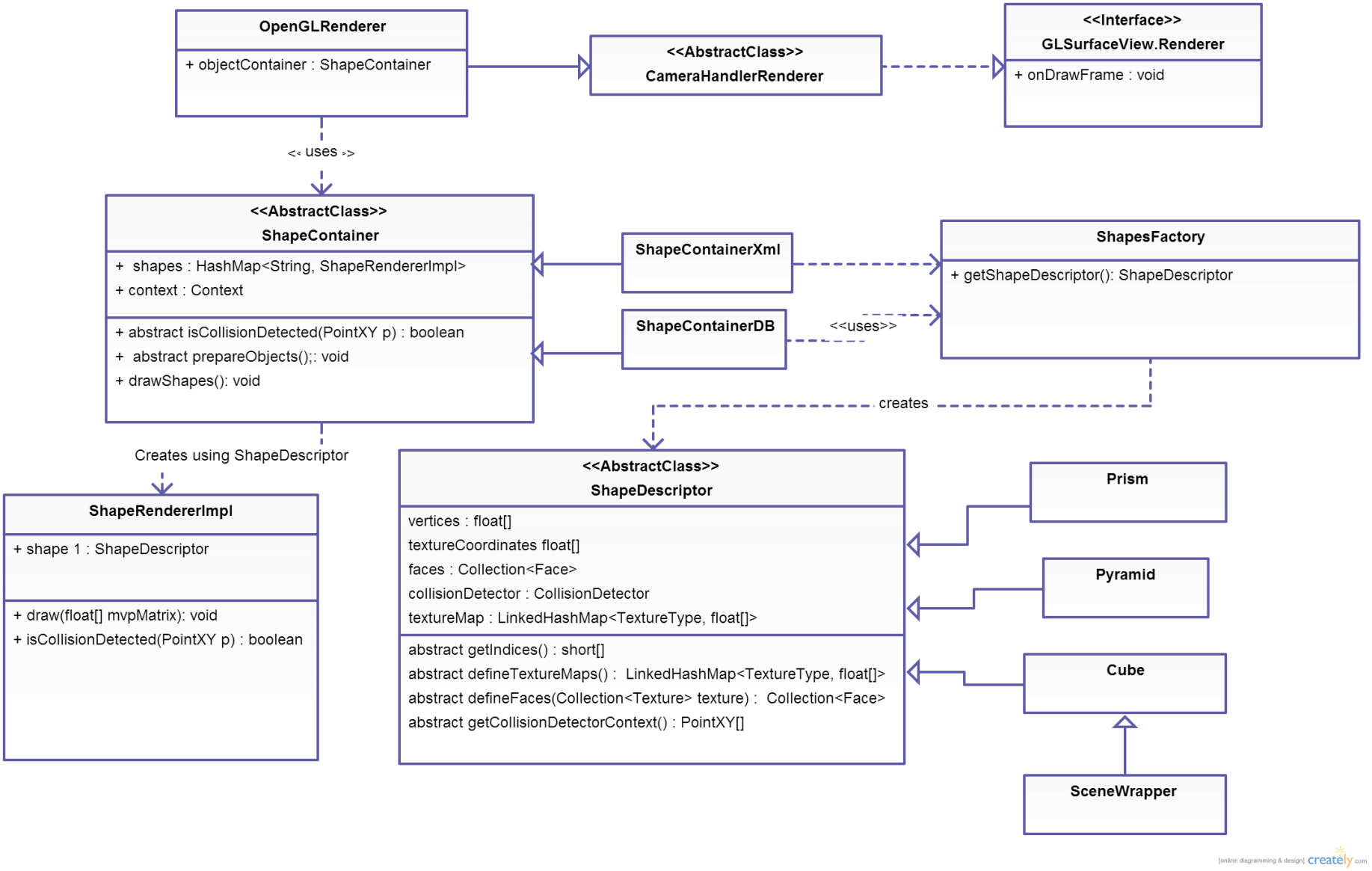
```
precision mediump float;
uniform sampler2D u_texId;
varying vec2 v_texCoords;
void main(){
        gl_FragColor = texture2D(u_texId, v_texCoords);
}
```

Jedini ulazni parametar programa za sjenčanje točaka je *u_texId.* On predstavlja 2D teksturu koja se treba pridružiti točki. Atribut *v_texCoords* je proslijeđen iz programa za sjenčanje vrhova te zajedno s *u_texId* predstavlja ulazne parametre za metodu *texture2D.* Metoda *texture2D* vrati boju točke kao vec4. Vraćeni rezultati se postave kao *gl_FragColor*, odnosno kao izlazna vrijednost programa za sjenčanje točaka.

4.1.1.10. Apstraktni kontejner za objekte

U prethodnim poglavljima u detalje je objašnjeno kako pojedini dijelovi aplikacije funkcioniraju, a u ovom poglavlju je cilj napraviti rezime svega i objasniti kako su pojedine stvari povezane u cijelinu.

Render klasa se brine oko iscrtavanja svih objekata i samo iscrtavanje se odrađuje u implementaciji metode *onDrawFrame()*. S obzirom da model ima više objekata koje treba prikazati, implementiran je kontejner koji sadrži sve objekte, te se pozivom jedne metode pozove metoda *draw()* na svim objektima. Kako bi to bilo što jednostavnije za shvatiti, na slici 23. je prikazan dijagram klasa za dio projekta koji to omogućuje. Bitno je napomenuti da dijagram klasa nije potpun, tj. prikazuje samo metode i atribute koji su relevantni za ovaj dio priče.



Slika 23 Dijagram klasa kontejner

U prvom redu su tri klase, odnosno podklase R*enderer* klase iz razloga što *OpenGlRenderer* koristi *ShapeContainer*. U *OpenGlRenderer.onSurfaceCreated()* metodi se inicijalizira *ShapeContainer*, tj. inicijalizira se podklasa *ShapeContainerDB* s parametrima *City* i *Context,* te se pozove metoda *prepareObjects()* kao što je prikazano u sljedećem okviru.

```
ShapeContainer  objectContainer = new ShapeContainerDB(getCity(), context);
objectContainer.prepareObjects();
```

Metoda *prepareObjects()* iz baze prikupi sve potrebne informacije, te pomoću *ShapesFactory.getShapeDescriptor()* kreira odgovarajući *ShapeDescriptor*. *ShapesFactory* na temelju *ShapeType* odlučuje koju implementacija *ShapeDescriptora* se kreira. Klasa *ShapesFactory* je prikazana u sljedećem okviru.

```
public class ShapesFactory {
        ShapeDescriptor shapeDescriptor;
        public ShapesFactory(ShapeType shapeType, Collection<Vertex> vertices,
Collection<Texture> textures){
                if (ShapeType.CUBE.equals(shapeType)){
                        shapeDescriptor = new Cube(vertices, textures);
                }else if(ShapeType.PRISM.equals(shapeType)){
                        shapeDescriptor = new Prism(vertices, textures);
                }else if(ShapeType.PYRAMID.equals(shapeType)){
                        shapeDescriptor = new Pyramid(vertices, textures);
                }else if(ShapeType.SCENE_WRAPPER.equals(shapeType)){
                        shapeDescriptor = new SceneWrapper(vertices, textures);
                }
        }
        public ShapeDescriptor getShapeDescriptor() {
                return shapeDescriptor;
        }}
```

ShapeDescriptor služi za opis objekta, pod time se misli na podklase ShapeDescriptor-a. ShapeDescriptor, na temelju podataka dobivenih iz baze (npr kolekcija Vertex i Textures klasa), pripremi podatke za OpenGL. U tablici 1. su prikazani podaci koje ShapeDescriptor

sadržava nakon što je obrada uspješno završena. Potrebno je napomenuti da se indeksi, koordinate vrhova i koordinate tekstura direktno prosljeđuju OpenGl-u bez daljnje obrade, za razliku od kolekcije *Face* klasa.

Tabela 1. *ShapeDescriptor*

Naziv	Tip	Svrha	Minimalan set podataka
Indices	*short[]*	Definira kako se trokuti renderiraju prema podacima iz *Vertices*	3 indeksa za jedan trokut, 2 trokuta za jednu stranicu (ako je četverokutna stranica).
Vertices	*float[]*	Sadrži informacije o koordinatama vrhova objekta.	Tri koordinate po vrhu.
Textures	*float[]*	Sadrži informacije o koordinata tekstura	Dvije koordinate po vrhu
Faces	*Collection<Face>*	*Face* je descriptor za jednu stranicu objekta.	Jedan *Face* po stranici

ShapeDescriptor sadrži zajedničke metode svih njenih podklasa i ograničava njene podklase da moraju implementirati metode kojim se opisuje svaki od objekata. To su sljedeće metode:

- *short[] getIndices()*
- *LinkedHashMap<TextureType, float[]> defineTextureMaps()*
- *Collection<Face> defineFaces(Collection<Texture> texture)*
- *PointXY[] getCollisionDetectorContext()*

Najprije će biti objašnjeno kako *ShapeDescriptor* poziva te metode i gdje se one koriste.

U klasi *ShapesFactory* se može vidjeti da se npr. klasa *Cube* inicijalizira s dva parametra *Collection<Vertex> vertices* i *Collection<Texture> textures. Cube* konstruktor poziva konstruktor od nadklase, odnosno konstruktor *ShapeDescriptor* klase s istim parametrima i parametrom *int[] verticesConvertionRules.* Na primjeru je prikazan konstruktor *ShapeDescriptor* klase.

```
public ShapeDescriptor(Collection<Vertex> vertex, Collection<Texture> textures, int[]
verticesConvertionRules) {
        vertices = convertVertices(vertex, verticesConvertionRules);
        faces = defineFaces(textures);  //abstract method
        textureMap = defineTextureMaps(); //abstract method
        textureCoordinates = defineTextureCoordinates(textures);
}
```

Metoda *convertVertices*

Metoda *convertVertices* koristi se da kolekciju *Vertex* klasa konvertira u listu *float* vrijednosti. Postoje dvije vrste konverzije, s i bez *verticesConvertionRules.* Verzija bez *verticesConvertionRules* konvertira na način da iterira po kolekciji i slijedno u listu dodaje koordinate x, y i z. Druga verzija implementacije zahtjeva listu pravila kao na primjeru ispod.

```
verticesConvertionRules = { 0, 1, 2, 3, 4, 5, 6, 7, 5, 3, 2, 6, 4, 7, 1, 0, 7, 6, 2, 1, 4, 0, 3, 5 }
```

U ovom slučaju kolekcija se prvo konvertira u listu kao u prvom primjeru, ali se nakon toga prema pravilima sortiraju vrhovi objekta. Konverzija s pravilima služi kako bi se iz liste vrhova dobio model koji odgovara OpenGL-u. Primjer iznad se koristi u *Cube* implementaciji i na temelju tog ulaza, zajedno s osam vrhova poligona, kreira listu s trideset i šest vrhova.

Metoda *defineFaces(Collection<Texture> texture)*

Metoda *defineFaces(Collection<Texture> texture)* je apstraktna metoda te se mora implementirati u nekoj od podklasa. Metoda vraća kolekciju *Face* klasa. Na primjeru ispod je prikazana implementacija metode *defineFaces()* koja kreira samo jednu stranicu.

```
@Override
protected Collection<Face> defineFaces(Collection<Texture> textures) {
        Collection<Face> faces = new ArrayList<Face>();
        Face face = new Face();
```

```
        face.setNumVertices(6);
        face.setOffset(12);
        face.setTexture(findTextureByFace(TextureType.FRONT, textures));
        faces.add(face);
        return faces;
}
```

Klasa *Face* ima atribute prikazane u tablici 2. koje je potrebno postaviti kako bi OpenGL mogao iscrtati tu stranicu.

Tabela 2. Atributi klase *Face*

Atribut	Tip	Opis
numVertices	*Int*	Broj vrhova kojim je definirana stranica objekta
offset	*Int*	Vrijednost koja se dobije umnoškom *numVertices* i veličinom u bajtovima tipa podataka koji opisuje indekse (tip *short*)
texture	*Texture*	Tekstura koja se koristi za ovu stranicu

Metoda *defineTextureMaps*

Metoda *defineTextureMaps* služi da se svakom tipu teksture (*enum TextureType)* dodijele adekvatne koordinate, kako bi se u metodi *defineTextureCoordinates*, za svaki *Face*, mogle dodijeliti prave koordinate. Na okviru ispod je prikazana implementacija u kojoj se dodaju koordinate tekstura za prednju i stražnju stranu objekta. Koristi se posebna vrsta *HashMap-a* koja osigurava postojanost redoslijeda umetanja objekata.

```
private static final float[] texCoordFront = { 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f };
private static final float[] texCoordBack = { 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f };
@Override
protected LinkedHashMap<TextureType, float[]> defineTextureMaps() {
        LinkedHashMap<TextureType, float[]> ret = new LinkedHashMap<TextureType,
float[]>();
        ret.put(TextureType.FRONT, texCoordFront);
        ret.put(TextureType.BACK, texCoordBack);
        return ret;
}
```

Metoda *defineTextureCoordinates()*

Metoda *defineTextureCoordinates()* na temelju proslijeđene kolekcije tekstura i prethodno postavljenih koordinata za svaki tip *TextureType* vrati listu koordinata tekstura. U obzir uzima i atribute *repeatTexture* kojima se definira da li se neka tekstura mora ponavljati.

Metoda getIndices()

Metoda getIndices() mora biti implementirana u konkretnoj klasi te vraća listu indeksa koji se koriste u *ShapeRendererImpl* klasi.

Metoda getCollisionDetectorContext()

Metoda getCollisionDetectorContext() služi da se definira bazna stranica, na temelju koje se računa detekcija sudara. Bazna stranica je definirana vrhovima lika koji opisuju tu stranicu. *Metoda getCollisionDetectorContext()* vraća listu objekata *PointXY* koji sadrži atribute x i y koji označavaju koordinatu x i koordinatu y za taj vrh. Koordinata z nije definirana jer se detekcija sudara računa u dvodimenzionalnom sustavu xy, te bi u ovom kontekstu bila redundantna. U okviru ispod je prikazana implementacija ove metode u *Cube klasi.*

```
@Override
protected PointXY[] getCollisionDetectorContext() {
        return new PointXY[] {
                new PointXY(vertices[0], vertices[1]),
                new PointXY(vertices[12], vertices[13]),
                new PointXY(vertices[21], vertices[22]),
```

```
            new PointXY(vertices[3], vertices[4])
        };
}
```

Nakon što je S*hapeDescriptor* kreiran, *ShapeContainer* kreira objekt tipa *ShapeRendererImpl* i dodaje ga u mapu s ostalim *ShapeRendererImpl* objektima kao što je prikazano na primjeru.

```
ShapeRendererImpl renderer = new ShapeRendererImpl(shapeDescriptor, context);
shapes.put(shape.getName(), renderer);
```

Klasa *ShapeRendererImpl*

Klasa *ShapeRendererImpl* na temelju informacija u *ShaderDescriptor-u* iscrtava model. Klasa *ShapeRendererImpl* sadrži metodu *draw(float[] mvpMatrix)* koja je zadužena za iscrtavanje objekta i poziva se iz *ShapeContainer.drawShapes()* metode na svim objektima koji se nalaze u tom kontejneru.

Prilikom inicijalizacije klase *ShapeRendererImpl* odrade se sljedeće akcije:

- Učita se program pomoću GlUtils.loadProgram (opisano u pogavlju 4.1.1.8)

```
programId = GlUtils.loadProgram(VertexShaders.getCubeVertexshadercode(),
FragmentShaders.getCubefragmentshadercode());
```

- Inicijaliziraju se međuspremnici za koordinate vrhova, koordinate tekstura i indekse
  Primjer za inicijaliziranje međuspremnika za koordinate vrhova

```
vertexBuffer = ByteBuffer.allocateDirect(shape.getVertices().length *
4).order(ByteOrder.nativeOrder()).asFloatBuffer();
vertexBuffer.put(shape.getVertices()).position(0);
```

- Učitaju se teksture pomoću metode *loadTexture*
  Metoda *laodTexture* iterira po svim *Face* objektima koje dohvati iz *ShapeDescriptora*, pomoću *GLUtils* i putanje u *Texture* objektu za taj *Face* kreira teksturu i postavlja identifikator kreiranog kontejnera kao vrijednost atributa *textureId* u objektu *Texture.*

```
Collection<Face> faces = shape.getFaces();
for (Face face : faces) {
        Texture texture = face.getTexture();
```

```
            String path = texture.getPath();
            int createTexture = GlUtils.createTexture(path, context);
            face.getTexture().setTextureId(createTexture);
}
```

- Rezerviranju GPU (engl. *Graphics Processing Unit*) resursi

  Prethodno kreirane međuspremnike s koordinatama je potrebno proslijediti na GPU kako bi podaci bili dostupni programima za sjenčanje. Rezerviranje resursa se izvodi kroz tri koraka:

  1. Generiranje međuspremnika (međuspremnik još nije alocirao memoriju);

```
GLES20.glGenBuffers(1, vbo, 0);
```

  2. Dohvaćanje kreiranog međuspremnika (ujedno i alocira memoriju);

```
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, vbo[0]);
```

  3. Proslijeđivanje podataka – koristi se prethodno kreirani međuspremnik

```
GLES20.glBufferData(GLES20.GL_ARRAY_BUFFER, vertexBuffer.capacity() *
BYTES_PER_FLOAT, vertexBuffer, GLES20.GL_STATIC_DRAW);
```

Metoda *draw(float[] mvpMatrix)*

Prilikom inicijalizacije *ShapeRendererImpl* klase alocirani su resursi na *GPU* i proslijeđeni su svi podaci potrebni da se iscrta objekt. Nakon inicijalizacije objekt još nije iscrtan, nego se iscrta tek nakon poziva metode *draw(float[] mvpMatrix*). Metoda *draw()* prima jedini parametar, matricu koja je kreirana u *OpenGLRender* klasi kao umnožak projekcijske matrice i matrice kamere. Matrica *mvpMatrix* će biti proslijeđena u program za sjenčanje u ovoj metodi.

Metoda *draw* izvodi sljedeće:

- Dohvaća program

```
GLES20.glUseProgram(programId);
```

- Dohvaća reference na ulazne parametre program za sjenčanje vrhova

```
iPosition = GLES20.glGetAttribLocation(programId, "a_position");
iVPMatrix = GLES20.glGetUniformLocation(programId, "u_VPMatrix");
iTexLoc = GLES20.glGetUniformLocation(programId, "u_texId");
iTexCoords = GLES20.glGetAttribLocation(programId, "a_texCoords");
```

- Proslijedi vrijednosti na ulazne parametre programa za sjenčanje vrhova (iz globalne memorije na GPU)
    - Koordinate vrhova

```
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, vbo[0]);
GLES20.glEnableVertexAttribArray(iPosition);
GLES20.glVertexAttribPointer(iPosition, 3, GLES20.GL_FLOAT, false, 0, 0);
```

    - Koordinate tekstura

```
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, tbo[0]);
GLES20.glEnableVertexAttribArray(iTexCoords);
GLES20.glVertexAttribPointer(iTexCoords, 2, GLES20.GL_FLOAT, false, 0, 0);
```

    - Matricu

```
GLES20.glUniformMatrix4fv(iVPMatrix, 1, false, mvpMatrix, 0);
```

- Koristeći indekse i kreirane teksture iscrtava elemente (objašnjeno u daljnjem tekstu)

Kako bi se iscrtali elementi sa zadanim teksturama potrebno je prvo dohvatiti međuspremnik s indeksima i postaviti aktivnu teksturu na *GLES20.GL_TEXTURE0*. To se postiže na sljedeći način:

```
int offset = 0;
GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, ibo[0]);
GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
```

S obzirom da objekt ima n stranica, potrebno je iterirati *Face* objekte u *ShapeDescriptoru*, te svaku stranicu iscrtati na način prikazan u sljedećem primjeru.

```
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, face.getTexture().getTextureId());
```

```
GLES20.glDrawElements(GLES20.GL_TRIANGLES, face.getNumVertices(),
GLES20.GL_UNSIGNED_SHORT, offset);
offset += face.getOffset();
```

Prva linija koda dohvati kontejner koji sadrži teksturu kreiranu prilikom inicijalizacije *ShapeRendererImpl.* Druga linija koda iscrta elemente, a zadnja linija povećava *offset. Offset* se koristi kako bi se postavio pokazivač na sljedeću lokaciju, na kojoj su indeksi za sljedeći trokut koji će se iscrtati.

4.2. Serverska aplikacija

Serverska aplikacija služi za posluživanje XML opisa koji definiraju model koji će se na Android aplikaciji iscrtati. Korištenjem takvih opsa se omogućuje prikaz različitih gradova te aplikacija nije ograničena eksplicitnim podacima i nije ograničena na prikaz samo jednog grada, trga i sl. Ova aplikacija nije u potpunosti implementirana, s time da ima jako puno mogućnosti za nadogradnju. Nije implementirana u potpunosti jer nije usko vezana uz temu diplomskog rada, ali je bila nužna da omogući centralizaciju konfiguracija i da se izbjegne kodiranje samog modela u izvornom kodu. Kao prvo, ne koristi se nikakav oblik baze, već su potrebni podaci eksplicitno zadani. Potrebni podaci su putanje do direktorija unutar kojeg se nalazi konfiguracija i sve potrebne datoteke za prikaz grada. Konfiguracije su u obliku XML datoteka i sadrže točan opis pojedinog objekta s njegovim koordinatama i teksturama, te putanju direktorija sa slikama koje će se koristiti kao teksture prema određenim pravilima. Također, direktorij za slike i slike se nalaze unutar istog direktorija.

4.2.1. Implementacija

Serverska aplikacija napisana je u Java programskom jeziku. Korištena je *RestEasy* biblioteka koja omogućuje jednostavno kreiranje web servisa koristeći anotacije. Više o samoj biblioteci će biti objašnjeno kroz cijelo poglavlje. Aplikacija se pokreće na Tomcat serveru i u OpenShift okruženju. OpenShift je otvorena platforma, pomoću koje je aplikacija instalirana na javnom repozitoriju. Na taj su način serverska aplikacija, odnosno podaci sa serverske aplikacije dostupni svim korisnicima Android aplikacije.

U nastavku ovog poglavlja je detaljno opisan svaki korak koji je bio potreban za kreiranje serverske aplikacije.

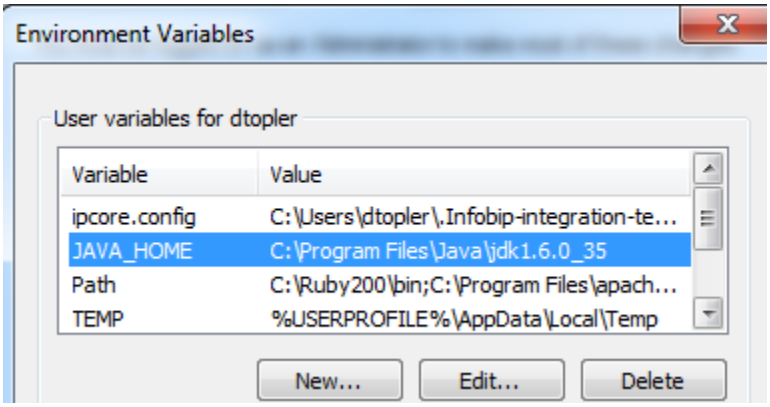
4.2.1.1.Priprema radnog okruženja

Prvi korak je priprema radnog okruženja. Pri tome se misli na instalaciju svih potrebnih alata koji su potrebni za razvoj serverske aplikacije. Potrebno je napomenuti da je sve alate treba instalirati kako bi se aplikacija uspješno pokrenula, te da se aplikacija razvijala na Windows 7 operativnom sustavu.

Java Development Kit (JDK)

Prvi korak u pripremi radnog okruženja je instalacija JDK-a. Za razvoj serverske aplikacije je korišten JDK verzije 1.6, koju je moguće preuzeti sa stranice *www.oracle.com*.

Po završetku instalacije potrebno je dodati putanju do instaliranog direktorija u sistemske varijable. Putanje je moguće postaviti na sljedeći način: *System -> Advanced system settings -> Environment variables*.



Slika 24. Dodavanje sistemskih varijabli

Eclipse

Serverska aplikacija je u potpunosti razvijena u Eclipse programskog okruženju. Eclipse je besplatni alat, odnosno IDE (engl. *Integrated development environment*) koji ima podršku za razvoj Java web aplikacija.

Da bi se započeo rad s *Eclipse*-om potrebno je preuzeti *Juno* verziju *Eclipse IDE for Java EE Developers* sa stranice *www.eclipse.org/downloads*. Također potrebno je na računalu imati instalirani JDK 1.6.x (*Java Development Kit*).

Eclipse je dovoljno raspakirati iz arhive te je moguće pokrenuti alat. Nakon pokretanja potrebno je odabrati radni prostor u kojem će se spremati projekti.

Apache Maven

Apache Maven je koristan alat koji omogućuje jednostavno upravljanje projektom. Pod upravljanje projektom misli se na sve zavisnosti koje glavni projekt ima prema ostalim projektima. Npr. serverska aplikacija koristi *RestEasy* projekt, te je serverska aplikacija ovisna o *RestEasy* projektu. *Apache Maven* omogućuje da se u ovom slučaju *RestEasy* i ostali zavisni projekti preuzmu i budu dostupni glavnom projektu. Sve takve zavisnosti se definiraju kroz XML datoteku koja se pod nazivom *pom.xml* nalazi u korijenskom direktoriju projekta. Sve definirane zavisnosti se jednom komandnom linijom mogu preuzeti sa udaljenih repozitorija.

Da bi se *Apache Maven* mogao koristi potrebno je preuzeti direktorij s *Apache* stranice te ga raspakirati. Potrebno je dodati „MAVEN_HOME“ varijablu u sistemske postavke kao što je prethodno napravljeno za *Java Development Kit*.

Apache Tomcat

Apache Tomcat je implementacija otvorenog koda *JavaServlet*-a i *JavaServerPages* tehnologija. [tomcat.apache.org, (2005)] *Apache Tomcat* omogućuje da se web aplikacije mogu instalirati i pokretati na lokalnom serveru.

Potrebno je preuzeti zadnju verziju (*Tomcat* 7) sa stranice *http://tomcat.apache.org/*. Preuzeti sadržaj dovoljno je raspakirati i referencirati ga iz *Eclipse*-a. Referenciranje *Tomcat* servera iz *Eclipse* okruženja je moguće na sljedeći način. Klikom na *Window -> Preferences -> Server -> Runtime Environments* se otvara prozor u kojem je moguće dodati *Server runtime environment*. Potrebno je odabrati *Apache Tomcat* v7.0 te dati putanju do prethodno preuzetog direktorija.

4.2.1.2.Kreiranje web projekta

Potrebno je kreirati *Maven* projekt te mu postaviti određene postavke kako bi projekt bilo moguće instalirati na *Apache Tomcatu*. Najjednostavniji način (način koji ne zahtijeva nikakve dodatke za *Eclipse* ili sl.) je da se kreira *Java* projekt i u korijenskom direktoriju projekta se doda *pom.xml*. Datoteku je potrebno popuniti s informacijama o projektu (naziv, verzija i dr.) te dodati sve zavisne projekte.

Sljedeći korak kreiranja web projekta je postavljanje *Project Facets*. Do tih postavki je moguće doći kroz izbornik klikom na *Project -> Properties -> Project Facets* ili desni klik na projekt, pa klik *Properties -> Project Facets*.
Potrebno je označiti da projekt koristi *Dynamic Web Module*. *Dynamic Web Module* dodaje podršku za *Java Servlet API*, za generiranje dinamičnog web sadržaja.

Zadnji korak je konfigurirati *Deployment Assembly* u *Properties* prozoru (otvoren u prethodnom koraku). Potrebno je pridružiti zavisne projekte, odnosno njihove biblioteke na putanju na kojoj se oni nalaze. U slučaju da se koristi *Maven*, svi zavisni projekti se spremaju u lokalni *Maven* repozitorij.

4.2.1.3.Zavisni projekti

Već je objašnjeno kako instalirati i podesiti *Apache Maven*. U nastavku će biti objašnjeno kako će se *Maven* koristiti. Prvi korak je popisati sve ovisne projekte koji su potrebni za ovaj projekt, a to su:

- *Javax-servlet;*
- *Org.jboss.resteasy;*
- *Org.codehouse.jackson.*

Neki od projekata imaju više artefakata, tako da je svakog od njih potrebno dodati u *pom.xml* na sljedeći način:

```
<dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jaxrs</artifactId>
        <version>2.2.1.GA</version>
</dependency>
```

Nakon što se na taj način definiraju svi ovisni projekti, s *mvn eclipse:eclipse* naredbom se automatski konfigurira *Eclipse* projekt te preuzmu svi ovisni projekti.

4.2.1.4.Konfiguriranje web.xml

Web.xml je skup pravila i postavki (engl. *deployment descript*or) koje će se izvršiti prilikom instaliranja aplikacije. *Web.xml* je korišten za konfiguriranje sljedećih stvari:

- Postavljanje kontekst parametara

```
<context-param>
```

```
<param-name>appConfigLocation</param-name>
    <param-value>
        ${user.home}/.WS-dtopler/config/app.properties
 </param-value>
</context-param>
```

- Dodavanje slušača za upravljanje konfiguracijskom datotekom

```
<listener>
        <listener-class>org.dip.config.ConfigListener</listener-class>
</listener>
```

- Inicijalizacija *Rest Easy* servleta

```
<servlet>
      <servlet-name>resteasy-servlet</servlet-name>
      <servlet-class>
    org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
      </servlet-class>
      <init-param>
         <param-name>javax.ws.rs.Application</param-name>
         <param-value>
            org.dip.config.RestServicesConfig
         </param-value>
      </init-param>
</servlet>
```

- Preslikavanje URL-a

```
<servlet-mapping>
        <servlet-name>resteasy-servlet</servlet-name>
        <url-pattern>/*</url-pattern>
</servlet-mapping>
```

RestServicesConfig

RestServicesConfig služi da se navedu sve klase koje imaju neku od *RestEasy* anotacija. To se izvede tako da se u set podataka dodaju instance svih menadžer klasa. Metode u menadžer klasi, koje se žele izložiti prema korisnicima, moraju imati neku od *RestEasy* anotacija koje će definirati postavke za izlaganje te metode.

```
@GET
@Path("{id}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_JSON})
public Response getCityByIdJson(@PathParam("id") String id) throws Exception{
        return Response.status(200).entity(getObjectMapper()
.writeValueAsString(getCityDataService(). findById(id))).build();
}
```

Navedena metoda koristi tri anotacije iz *RestEasy* biblioteke. *@GET* definira da će samo HTTP GET zahtjevi moći pozvati ovu metodu. *@Path* određuje preslikavanje metoda na neku putanju, s napomenom da se putanja dodaje na putanju na koju je cijela klasa preslikana. *@Produces* služi da pridruži klijentski *Accept header*. U ovom slučaju će HTTP odgovor sa servera sadržavati *Content-type : application/json*. Metoda prima parametar *id* koji će se preslikati iz putanje te kao odgovor šalje *json* serializirani objekt koji se dobije pozivom metode *findById* na objektu tipa *cityDataService*. *DataService* upućuje da je riječ o servisu koji zna komunicirati s bazom. U ovom slučaju, iznimno zbog jednostavnosti, jedina implementacija *cityDataService*-a ne dohvaća podatke preko baze, već su ti podaci ekspilicitno zadani.

4.2.1.5.Instalacija servisa na OpenShift

Postoje dvije verzije serverske aplikacije, osnovna i druga prilagođena za rad s OpenShift platformom. Glavna razlika je što OpenShift verzija sadrži specifične konfiguracije koje omogućuju da se aplikacija pokreće i radi unutar *OpenShift Cloud*-a. OpenShift je platforma razvijena od Red Hat-a te se temelji na principu PaaS (*Platform as a Service*). Paas je način za iznajmljivanje hardvera, operacijskih sustava, spremnika i mrežnih kapaciteta preko Interneta. [searchcloudcomputing.techtarget.com, 2010] OpenShift ima besplatan paket koji nudi ograničeni spremnik i ograničen broj aplikacija koje se mogu instalirati na njihovoj platformi. Nudi tri različita načina integracije: kroz web sučelje, kroz komandnu liniju i kroz razvojni

alat. Za integraciju programskog rješenja korištena su prva dva načina. Prvi način omogućuje da se preko web sučelja kreira projekt određenog tipa te je potom preko Git sustava za verzioniranje moguće ažurirati sadržaj na OpenShift repozitoriju. Drugi način, odnosno integracija kroz komandnu liniju, omogućuje korištenje Git sustava za verzioniranje kroz komandu liniju, ali nudi dodatne mogućnosti koristeći OpenShift-ov klijentski alat. Klijentski alat se naziva *rhc* te omogućuje upravljanje aplikacijom (npr. ponovno pokretanje, stopiranje i sl.). Također, omogućuje korištenje sustava za kontinuiranu integraciju (engl. *continuous integration system*) – *Jenkinsa. Jenkins* omogućuje jednostavno i automatizirano kreiranje verzija programskog rješenja, uz okruženje za automatsko pokretanje testova i sl. [wiki.jenkins-ci.org, 2005]

4.2.2. Način korištenja

Razlog kreiranja servisa je ideja da se svi opisi objekata na temelju kojih će se kreirati trodimenzionalni objekti u OpenGL-u, izdvoje na jedno mjesto, odnosno udaljeni repozitorij. Dakle, pod udaljeni repozitorij u kontekstu Android aplikacije, misli se na skup konfiguracija koje ovaj servis poslužuje.

Kako bi servis imao neku svrhu, potrebno je konfiguracije spremiti na datotečni sustav koji je dostupan servisu. Te konfiguracije će biti dostupne kroz izloženo aplikacijsko programsko sučelje.

4.2.2.1.Konfiguriranje servisa

Da bi servis mogao posluživati dodane konfiguracije, potrebno je konfigurirati i sam servis. Servis očekuje da je konfiguracija spremljena na putanji *${user.home}/.WS-dtopler/config/app.properties*, gdje je *${user.home}* početni korisnički direktorij na operativnom sustavu (npr. na *Windows* 7 operativnom sustavu i s prijavljenim korisnikom *dtopler* taj direktorij je *C:\Users\dtopler*).

U *app.properties* se mora definirati putanja datotečnog sustava do korijenskog direktorija gdje se nalazi repozitorij s konfiguracijama.
Primjer:
storage.path=C:\\Users\\dtopler\\Desktop\\StorageTest

4.2.2.2.Pokretanje servisa

Prethodno je objašnjeno kako instalirati *Apache Tomcat* u *Eclipse* okruženju. Potrebno je još nekoliko narednih koraka da bi se aplikacija pokrenula:

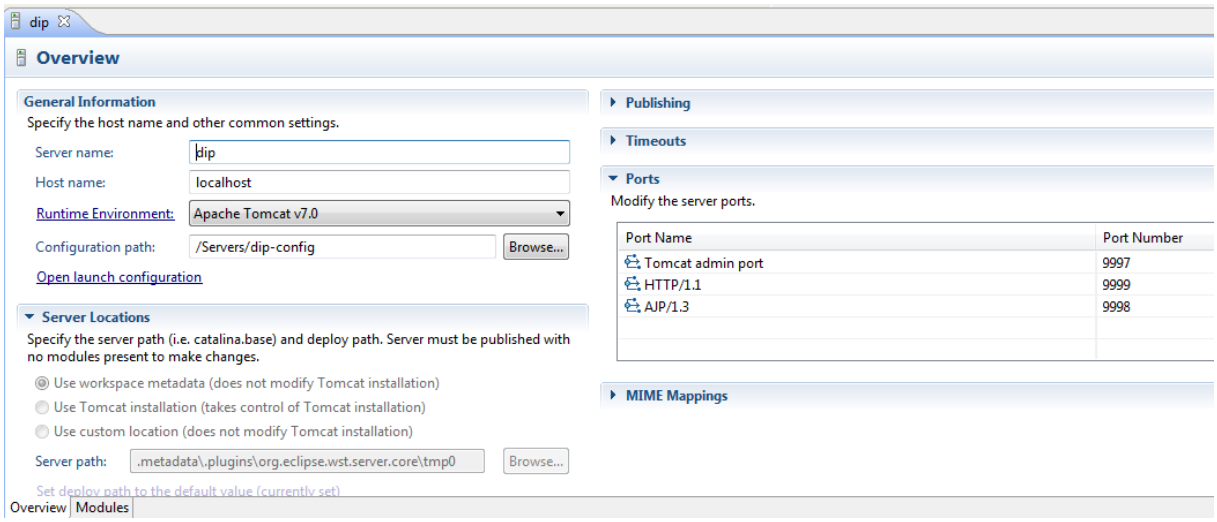
- Dodati *Tomcat* server;
- Konfigurirati server;
- Instalirati aplikaciju;
- Pokrenuti server.

Dodavanje *Tomcat Servera*

U *Eclipse* okruženju klikom na *Window -> Show view -> Servers* otvara se prozor s kreiranim serverima. Na desni klik *New -> Server* dodaje se novi server nakon postavljanja osnovnih informacija o serveru kao naziv, verzija servera i sl.

Konfiguriranje *Tomcat Servera*

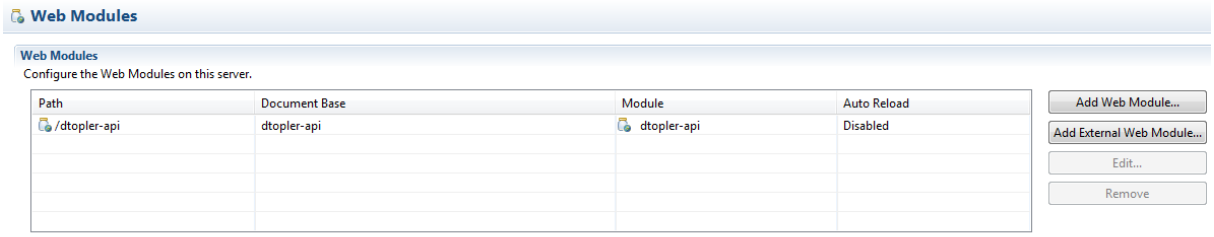
Dvostrukim klikom na dodani server otvaraju se u novoj kartici podaci s osnovnim informacijama o serveru kao na slici 36. Najvažnije je podesiti *portove* na one za koje znamo da nisu zauzeti.



Slika 25. Konfiguriranje *Tomcat* servera

Instaliranje modula na *Tomcat Server-u*

Klikom na *Modules* u kartici s osnovnim informacijama o serveru (slika 36.) otvara se kartica kao na slici 37.



Slika 26. Instaliranje Tomcat modula

Klikom na *Add Web Module*.. se mogu odabrati neki od postojećih modula u radnom okruženju. Web module se kreirao kada je dodan *Project Facets* kao u poglavlju 4.2.1.2.
Na slici 37. je prikazano stanje nakon što je modul već dodan.

Startanje *Tomcat* servera

Nakon što je *Tomcat* konfiguriran startanje servera je trivijalno. Desnim klikom na dodani server otvori se izbornik u kojem možemo odabrati način na koji se želi startati server. Moguće opcije su običano pokretanje servera (engl. r*un*) ili mod za istraživanje grešaka (engl. *debug*).

4.2.2.3.Korištenje API-a

Aplikacijsko programsko sučelje se može koristiti samo određenim HTTP zahtjevima koji su definirani u tablici 3.

Tabela 3. Lista poziva na server API

Putanja	HTTP Metoda	Parametri	Opis
/city	GET	-	Vraća sve gradove koje vrati *cityDataService*
/city/{id}	GET	*Id* – parametar iz putanje, predstavlja identifikator grada koji se pretražuje	Vraća rezultat pretrage gradova prema definiranom parametru *id* ili *null* ako grad nije pronađen.

/stream	GET	*Path* – predstavlja putanju do datoteke koju se želi preuzeti	Omogućuje prijenos datoteke koja je na putanji definiranoj parametrom *path*.

4.2.2.4.Konfiguriranje spremnika

Spremnik je zamišljen kao direktorij s poddirektorijima, gdje je svaki poddirektorij spremnik za pojedini grad. Zbog jednostavnosti implementacije, spremnik za pojedini grad mora biti točno u sljedećem formatu:

- varazdin-hr - poddirektorij s nazivom identifikatora grada za koji su pohranjene konfiguracije (npr. varazdin-hr)
    - *icon.jpg* – ikona koja će se u aplikaciji prikazivati u izborniku za taj grad
    - identifikator.zip – *zip* arhiva sa sljedećom hijerarhijom
        - images – direktorij sa slikama za teksture
            - image1.jpg
            - image2.jpg
            - ...
        - *config.xml* – konfiguracijska XML datoteka

4.2.2.5.Pravila za kreiranje konfiguracijske XML datoteke

Konfiguracijska datoteka mora biti točno u odgovarajućem formatu, iz razloga da bi klijentska aplikacija znala pravilno odrediti odgovarajuće informacije za pojedine objekte.

Primjer konfiguracijske datoteke:

```
<modelDescription>
        <city>Varazdin</city>
        <description>Varazdin je grad na sjeveru Hr..</description>
        <modelSizeX>450</modelSizeX>
        <modelSizeY>650</modelSizeY>
        <shapes>
                <shape>
                        <name>foi</name>
```

```
                    <type>cube</type>
                    <vertices>
                        <vertex x="41.0" y="0.0" z="0" />
                    </vertices>
                    <textures>
                        <texture location="Front">
                            <path>varazdin-hr/images/empty.jpg</path>
                        </texture>
                    </textures>
                </shape>
            </shapes>
</modelDescription>
```

Prikazani model je samo predložak konfiguracije, a u tablici 4. slijedi opis svakog pojedinog elementa XML konfiguracije.

Tabela 4. Opis Konfiguracijske XML datoteke

Element	Roditelj	Atributi	Opis
modelDescription	-	-	Služi kao kontejner za ostale elemente
city	*modelDescription*	-	Identifikator grada koji je opisan kroz konfiguraciju
description	*modelDescription*		Opis grada

modelSizeX	*modelDescription*	-	Veličina modela po osi X
modelSizeY	*modelDescription*	-	Veličina modela po osi Y
shapes	*modelDescription*	-	Kontejner za *shape* elemente
shape	*shapes*	-	Predstavlja jedan objekt u OpenGL 3D modelu
name	*shape*	-	Identifikator objekta, mora biti jedinstven na razini modela
type	*shape*	-	Vrsta objekta. Detaljnije u nastavku teksta
vertices	*shape*	-	Kontejner za koordinate objekta
vertex	*vertices*	x – koordinata x, y – koordinata y, z – koordinata z	Svaki pojedini vrh objekta mora biti definiran s 3 koordinate (x, y, z)
textures	*shape*	-	Kontejner za opis tekstura

texture	*textures*	*location* – specificira kojoj stranici objekta pripada ova tekstura	Opis svake pojedine teksture.
path	*texture*	*repeatTexture* – definira koliko puta da se tekstura ponavlja	Putanja na kojoj se nalazi slika koja predstavlja teksturu

Textures

Pomoću *textures* kontejnera moguće je definirati koje stranice objekta će se prikazivati. U obzir se uzimaju samo stranice koje imaju pridruženu teksturu. U slučaju da određena stranica nema definiranu teksturu tada se ona zanemaruje i ne iscrtava.

Klijentska aplikacija prepoznaje sljedeće lokacije tekstura:

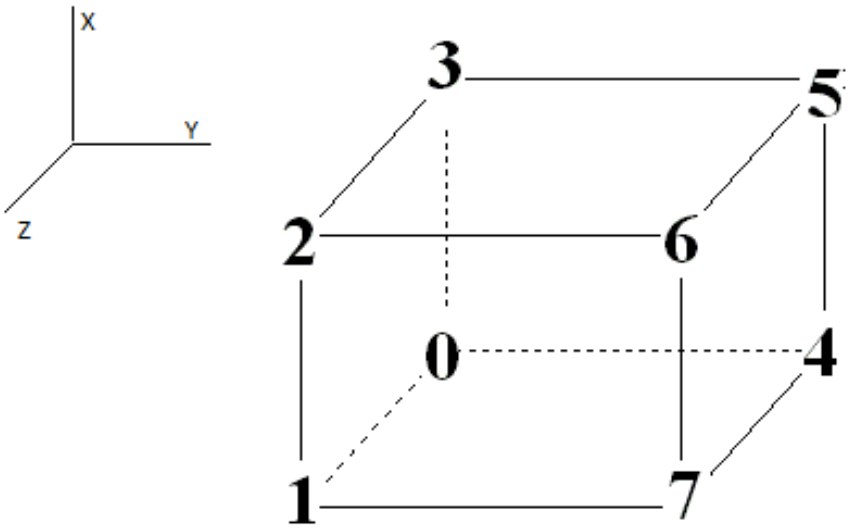
- *Front*
- *Back*
- *Left*
- *Right*
- *Top*
- *Bottom*

Type

Postoji nekoliko tipova koji su podržani u klijentskoj aplikaciji. Svaki od tih tipova ima različita svojstva, a samim time i razlike u načinu definiranja pojedinog. Tipovi su:

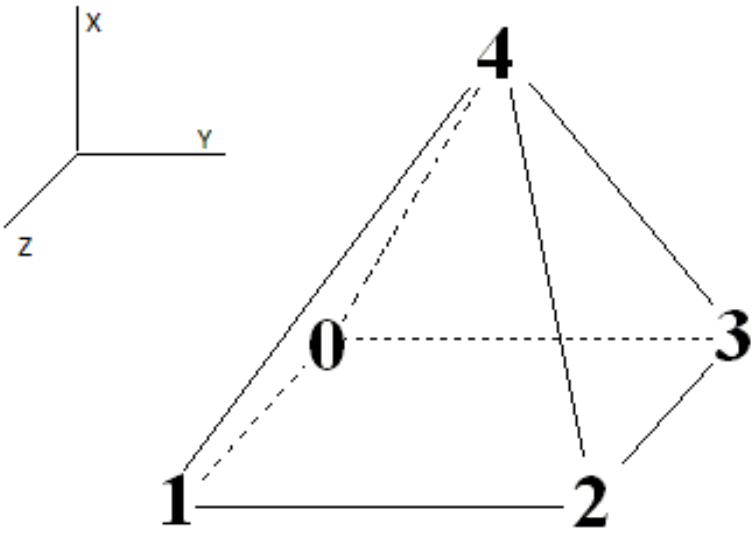
- *Cube*
- *SceneWrapper*
- *Pyramid*

Tip *Cube* označava objekt sa šest stranica i osam vrhova. Potrebno je u *vertices* kontejneru definirati svih osam vrhova kako bi se u OpenGL-u ispravno pridružili svi vrhovi, njihovi indeksi i koordinate tekstura. Na slici 38. je prikazano točno kojim redoslijedom vrhovi moraju biti definirani kako bi se dobro pridružile sve ostale postavke u klijentskoj aplikaciji.



Slika 27. Definiranje vrhova – *Cube*

Pyramid je tip koji definira objekt u obliku piramide koji ima pet vrhova i pet stranica. Također je potrebno definirati sve vrhove kako bi se piramida ispravno iscrtala. Na slici 39. je prikazan točan redoslijed kojim vrhovi moraju bit definirani.



Slika 28. Definiranje vrhova - *Pyramid*

SceneWrapper je tip koji je identičan tipu *Cube* s razlikom da se teksture (ako su definirane) iscrtaju s unutarnje strane. U klijentskoj aplikaciji ovaj tip ima svrhu samo za ograničavanje scene, odnosno iscrtavanje podne teksture i neba.

Vertices kontejner služi da se definiraju svi vrhovi svakog pojedinog objekta. Svaki vrh mora imati tri atributa (x, y, z). Objekt se iscrtava na definiranim koordinatama bez da se one dodatno manipuliraju.

4.3. Klijentska aplikacija

Klijentska aplikacija je nastala kao realizacija ideje da bilo koja aplikacija može koristiti izložene metode u serverskoj aplikaciji, bez da mora poznavati serversku aplikaciju, niti gdje je ona instalirana. Klijentska aplikacija je zapravo biblioteka koja služi jedino kao konektor prema serverskoj aplikaciji. Koristeći biblioteku, dovoljno je dodati je u projekt te se mogu koristiti metode koje sa serverske aplikacije dohvaćaju podatke. Jedina pretpostavka je da aplikacija ima pristup Internetu.

4.3.1. Implementacija

Aplikacija je *Java* aplikacija, s *pom.xml* datotekom u korijenskom direktoriju kako bi se definirali zavisni projekti. *Maven* plugin se ovdje koristi i kako bi se kreirala *jar* datoteka koja će biti ovisni projekt Android aplikaciji. *Jar* datoteka se kreira nakon što se izvrši naredba *mvn clean install* i spremi se u MAVEN_HOME direktoriju.

Osnovni dio aplikacije je klasa *Client*. Kada se ta klasa inicijalizira, inicijaliziraju se sve klase koju su tu registrirane i koje će biti dostupne drugim aplikacijama.

```
public class Client {
	private CityClient cityClient = null;
	private FileClient fileClient = null;
	public Client(ClientConfig clientConfig){
		this.cityClient = new CityClientImpl(clientConfig);
		this.fileClient = new FileClientImpl(clientConfig);
	}
	public CityClient getCityClient() {
		return cityClient;
	}
	public FileClient getFileClient() {
		return fileClient;
	}
}
```

CityClientImpl i *FileClientImpl* su klase koje znaju komunicirati sa serverskom aplikacijom, iz razloga jer proširuju *BasicClientImpl* klasu. *BasicClientImpl* sadrži metode koje komuniciraju sa serverskom aplikacijom. Klase koje proširuju *BasicClientImpl* koriste te metode na sljedeći način.

```
String url = getUrl(„/someUrl“)
HttpURLConnection connection = executeGet(url);
City city = deserialize(connection, City[].class);
```

U prvoj liniji se poziva metoda *getUrl(String url)* koja vraća cijelu putanju za pristup serveru. U drugoj liniji metoda *executeGet(String url)* otvara HTTP konekciju prema serveru, odnosno prema parametru *url*. Treća linija poziva metodu *deserialize(HttpURLConnection connection, Class<T> clazz)* koja vraća objekt tipa T. Primjer gore će vratiti objekt tipa *City[]* jer je drugi parametar *City[].class*. U *deserialize* metodi se iz konekcije čita HTTP status kod. Ako je status kod 200 onda se pročita *InputStream* i deserializiraju primljeni podaci u objekt tipa *City[]*. U sljedećoj tablici prikazane su sve metode koje ova aplikacija izlaže korisniku.

Tabela 5. Klijentske metode

Naziv interfejsa	Naziv metode	Parametri	Rezultat
CityClient	getAllAvaialbleCities	-	City[]
CityClient	getCityById	String id	City
FileClient	getStream	String id	DownloadDescriptor
FileClient	getStream	City city	DownloadDescriptor
FileClient	getStream	String url	DownloadDescriptor

DownloadDescriptor je klasa u kojoj su sve potrebne informacije za preuzimanje datoteke. Atributi su *InputStream* i veličina sadržaja.

4.3.2. Kako je koristiti?

Kako bi se ova biblioteka koristila potrebno je postaviti biblioteku u putanju (engl. *classpath*) projekta te inicijalizirati *Client* klasu. Jedini način za inicijalizaciju *Client* klase je preko konstruktora kojem je potrebno proslijediti *ClientConfig* instancu. *ClientConfig* je klasa koja služi za konfiguriranje klijenta. S obzirom da je riječ o veoma jednostavnom primjeru, sve što

je potrebno je putanja za pristup servisu. U primjeru je prikazano kako se inicijalizira *Client* s lokalnim postavkama.

```
ClientConfig clientConfig = new ClientConfig();
clientConfig.setUrl("http://192.168.5.102:9999/dtopler-api");
Client c = new Client(clientConfig);
```

U prethodnom poglavlju je objašnjeno da *Client* klasa sadrži instance svih modula koji znaju komunicirati sa server aplikacijom. Svaki od tih modula imaju *getter* u *Client* klasi koji omogućuje da se na sljedeći način dohvati modul, te pozove metoda za dohvat podataka.

```
City[] allAvaialbleCities = c.getCityClient().getAllAvaialbleCities();
```

Dohvaćeni podaci su pretvoreni u objekte tipa definiranog metodom u serverskoj aplikaciji. Kreirani objekt ima vrijednosti kojima se pristupa koristeći *get* metode implementirane u objektu.

Koristeći klijentsku aplikaciju pojednostavljuje se dohvaćanje podataka sa udaljenog servera. Korisniku su dostupna sučelja pomoću kojih dohvaća podatke i nema potrebe za implementacijom dodatnih funkcionalnosti, osim logike koja njegova aplikacija zahtijeva.

5. Korisnička dokumentacija

Android aplikaciju je moguće preuzeti na http://rg.c-hip.net/diplomski/Topler. Preuzeti sadržaj je *.apk* (engl. *application pacgake file*) datoteka, koju je potrebno pokrenuti kako bi se aplikacija instalirala. Nakon uspješnog pokretanja aplikacije korisnik dobije izbornik kao na slici 24., te su prikazane sljedeće opcije:

- Pregled gradova;
- Ažuriranje gradova;
- Postavke aplikacije.

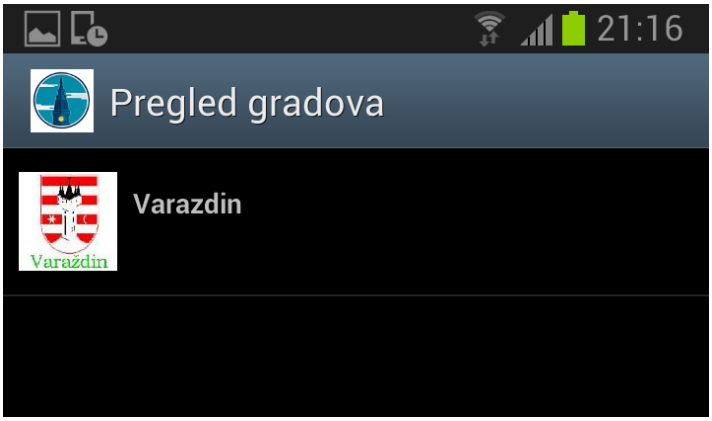
Svaka od opcija će biti detaljno objašnjena u nastavku.

Slika 29. Izbornik aplikacije

5.1. Pregled gradova

Ekran „Pregled gradova“ omogućuje korisniku da dobije listu svih gradova u lokalnom repozitoriju. Pod lokalni repozitorij misli se na sve konfiguracije i datoteke koje su preuzete s udaljenog repozitorija i spremljene u memoriju uređaja. Prilikom inicijalnog pokretanja aplikacije ova lista će biti prazna. Također, biti će prazna dok se neki od gradova ne ažurira. U nastavku slijedi opis ekrana „Ažuriranje gradova“.

Na slici 25. je primjer s jednim gradom u lokalnom repozitoriju. Odabirom grada otvara se novi ekran u kojem će se iscrtati model za taj grad prema konfiguracijama u lokalnom repozitoriju.

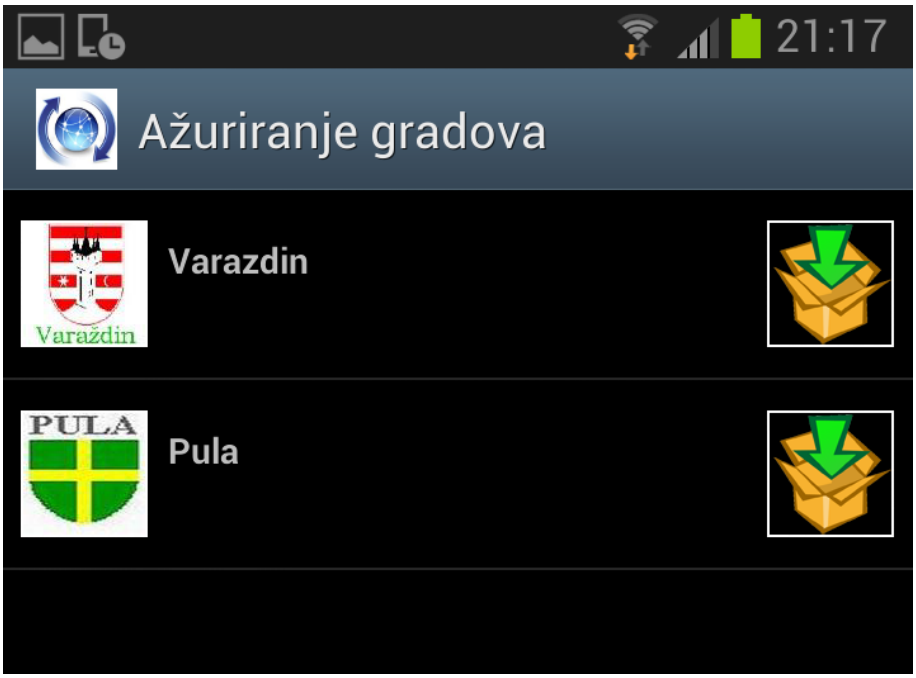


Slika 30. Pregled gradova

5.2. Ažuriranje gradova

Ekran „Ažuriranje gradova“ služi za ažuriranje postojećih gradova i dodavanje novih gradova u lokalni repozitorij. Za korištenje ove opcije potrebna je Internet konekcija kako bi uređaj mogao komunicirati s udaljenom serverskom stranom. Dohvaćanje liste gradova koji se mogu ažurirati se također odrađuje preko udaljenog servera.

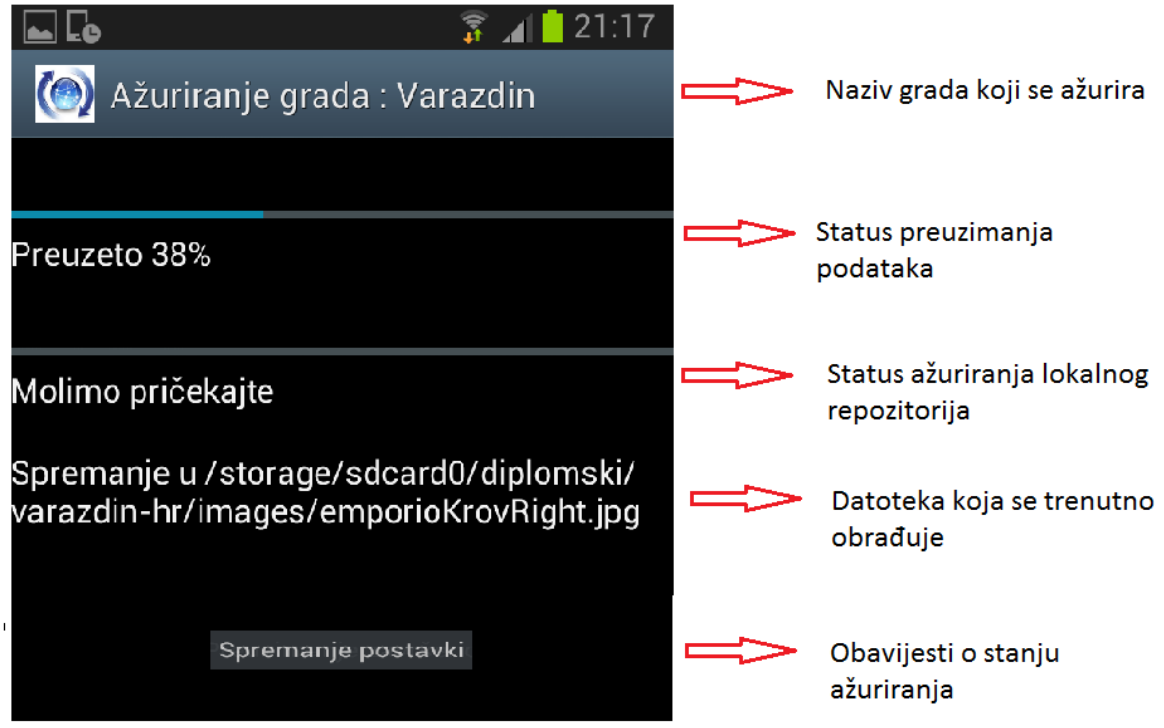
Na slici 26. je prikazan ekran za ažuriranje gradova. Uz svaki grad koji se nalazi na listi mogućih gradova je ikonica koja označava da li korisnik ima u lokalnom repozitoriju najnovije izmjene. Odabirom grada se otvara novi ekran u kojem je moguće pratiti stanje preuzimanja i ažuriranja lokalnih podataka. U nastavku slijedi detaljno objašnjenje ekrana za preuzimanje podataka.



Slika 31. Ažuriranje gradova

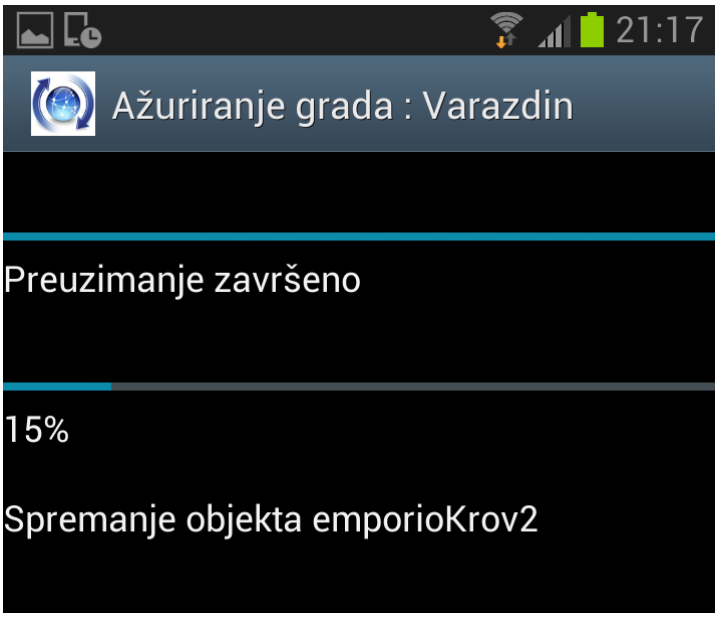
5.3. Ažuriranje grada

Odabirom grada iz liste dostupnih gradova za ažuriranje otvara se ekran „Ažuriranje grada“, te automatski započinje preuzimanje podataka iz serverske aplikacije prikazano kao na slici 27.



Slika 32. Ažuriranje grada – preuzimanje

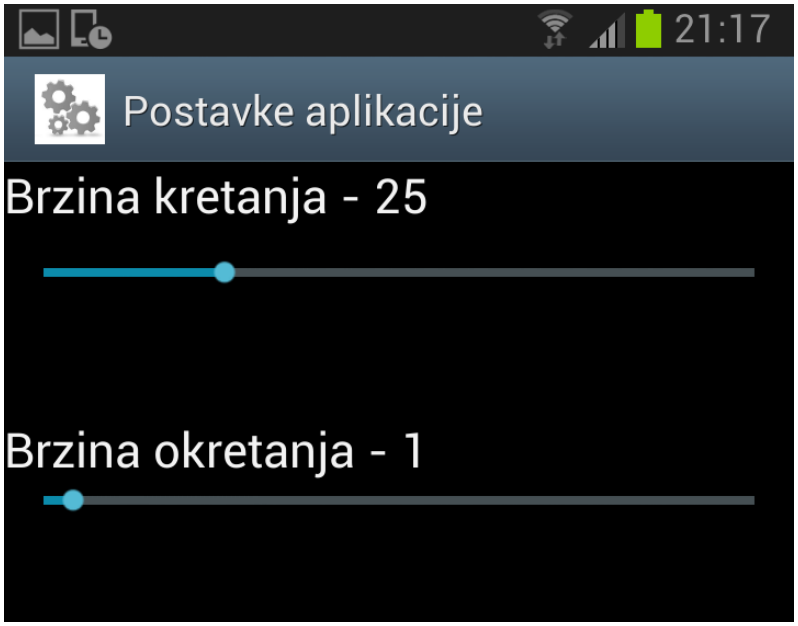
Nakon što završi preuzimanje podataka, započinje i ažuriranje lokalnog repozitorija. Ažuriranje lokalnog repozitorija je prikazano na slici 28. Na slici se vidi da je preuzimanje završeno, te da se upravo sprema objekt s nazivom „emporioKrov2“.



Slika 33. Ažuriranje gradova - spremanje

5.4. Postavke aplikacije

Ekran „Postavke aplikacije“ služi za konfiguriranje postavki brzine rotiranja i brzine kretanja. Postavke se odnose za ekran „Pregled gradova“ te se pomoću njih može konfigurirati i prilagoditi brzina kretanja na pojedinom uređaju. Na slici 29. je prikazan ekran na kojem je moguće konfigurirati navedene postavke.

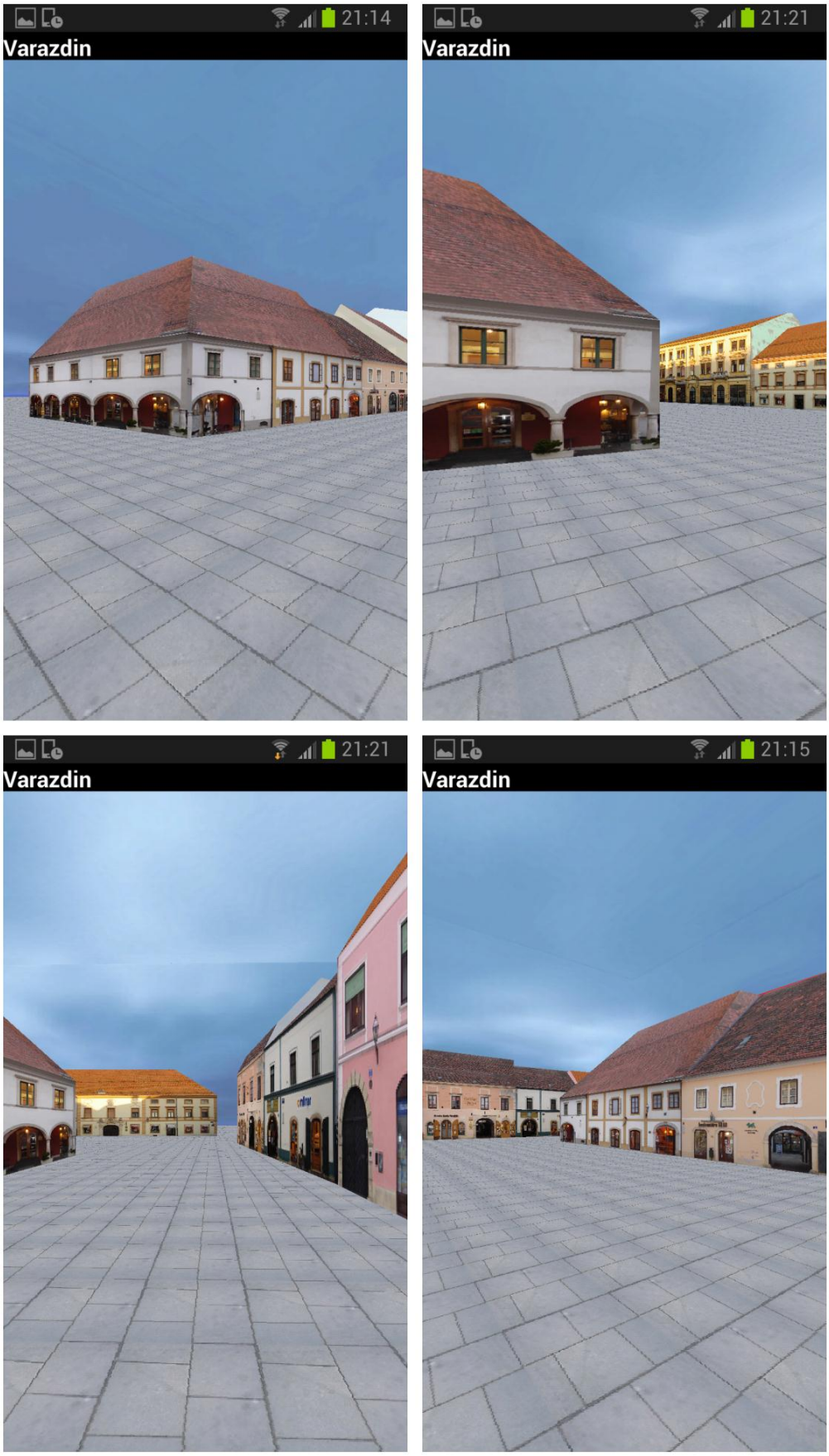


Slika 34. Postavke aplikacije

5.5. Pregled grada (3D model)

Na ekranu „Pregled gradova“ korisnik ima mogućnost odabira dostupnih gradova. Odabirom se otvara ekran u kojem će se iscrtati trodimenzionalni model. Za korištenje ove mogućnosti nije potrebno imati konekciju s udaljenim repozitorijom, jer su svi podaci preuzeti i spremljeni na samome uređaju. Na sljedećim slikama su prezentirani primjeri ekrana iz modela za grad Varaždin, odnosno Trg kralja Tomislava u Varaždinu.

Slika 35. Prikaz vijećnice



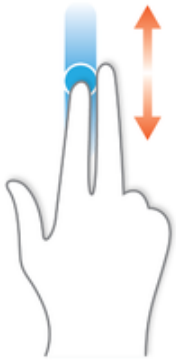
Slika 36. Prikaz varaždinskog korza iz više kutova

Slika 37. Prikaz Fakulteta organizacije i informatike

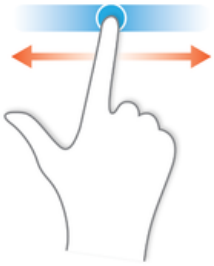
Korisniku je omogućena interakcija s modelom, odnosno omogućeno je kretanje po virtualnom trgu. Kretanje je omogućeno kroz dva aspekta:

- Kretanje u smjeru naprijed – natrag
- Okretanje lijevo – desno, tj. promjena smjera pogleda

Kretanje u smjeru naprijed – natrag omogućeno je kretnjom prikazanom na slici 33., dok je okretanje lijevo –desno omogućeno kretnjom prikazanom na slici 34.



Slika 38. Pomicanje s dva prsta (Two Finger Scroll) [kenfolios.com, 2013]



Slika 39. Okretanje (Flick Gesture) [kenfolios.com, 2013]

6. Zaključak

U ovom radu je prikazana izrada aplikacije koja sadrži interaktivnu 3D računalnu grafiku na Android mobilnoj platformi, koristeći standard OpenGL ES 2.0. Integracija Android platforme i OpenGL ES sustava je prikazana kroz rješenje programskog problema izrade interaktivnog 3D modela gradskog trga.

U prvim fazama razvoja programskog rješenja ideja je bila kreiranje aplikacije koja će prikazivati 3D model varaždinskog Trga kralja Tomislava. Daljnjim razvojem nastala je potreba da se podaci, koji definiraju objekte za iscrtavanje, izdvoje iz izvornog koda. Iz tog razloga, razvijen je sustav koji učitava XML datoteku i na temelju nje iscrtava sve objekte. XML datoteka sadrži opis modela koji je razvijen za potrebe ovog rada. Takvim pristupom razvoj aplikacije je odvojen u dva smjera. Prvi smjer je razvoj sustava za kreiranje i iscrtavanje 3D modela iz XML-a, a drugi obogaćivanje modela u vidu dodavanja preciznijih objekata i detaljnijih tekstura koristeći XML opise.

Razvijen je sustav, temeljen na klijent-server pristupu, kojim se riješio problem izdavanja nove verzije aplikacije prilikom svake promjene modela. Kako bi promjene XML opisa postale dostupne klijentima, potrebno ih je samo ažurirati na serveru. Također, aplikacija više nije ograničena na prikaz jednog varaždinskog trga, nego je omogućeno dodavanje proizvoljnih modela za druge trgove, gradove i slično.

Konačno, kvaliteta i brzina interaktivne 3D grafike koja je postignuta u aplikaciji (pogotovo kada se izvršava na novijim i moćnijim mobilnim uređajima, konkretno Samsungu Galaxy S3 na kojem je razvijana i testirana aplikacija) gotovo ne zaostaje za desktop računalima i iako je OpenGL ES na neki način samo podskup standardnom OpenGL-u ipak omogućuje potpuni doživljaj interaktivne 3D grafike na mobilnim uređajima, pogotovo u svojoj verziji 2.0.

7. Literatura

1. Ginsberg D, Munshi A, Shreiner D (2008) OpenGL ES 2.0 Programming Guide
2. Khronos Group, (2013a). Dostupno 20.8.2013. na http://www.khronos.org/
3. Khronos Group, (2013b), About The Khronos Group. Dostupno 20.8.2013. na http://www.khronos.org/about
4. Khronos Group, (2013c), The Standard for Embedded Accelerated 3D Graphics. Dostupno 20.8.2013. na http://www.khronos.org/opengles/
5. Khronos Group, (2013d), OpenGL ES 2.X. Dostupno 20.8.2013. na http://www.khronos.org/opengles/2_X/
6. Khronos Group, (2013e), Khronos Native Platform Graphics Interface. Dostupno 21.08.2013. na http://www.khronos.org/registry/egl/specs/eglspec.1.4.20130211.pdf
7. Simpson R, (2006), The OpenGL ES Shading Language. Dostupno 24.8.2013. na http://www.khronos.org/files/opengles_shading_language.pdf
8. qt-project.org, (2013), Cube OpenGL ES 2.0 example. Dostupno 16.7.2013. na http://doc-snapshot.qt-project.org/5.0/qtopengl/cube.html#loading-textures-from-qt-resource-files
9. opengl.org (2013), OpenGL Shading Language. Dostupno 20.7.2013. na http://www.opengl.org/documentation/glsl/
10. Multimedia Application Division, (2010), High-End 3D Graphics with OpenGL ES 2.0. Dostupno 21.8.2013. na http://www.freescale.com/files/dsp/doc/app_note/AN3994.pdf
11. developer.android.com, (2013a), Android, the world's most popular mobile platform. Dostupno 25.7.2013. na http://developer.android.com/about/index.html
12. developer.android.com, (2013b), Application fundamentals. Dostupno 25.8.2013. na http://developer.android.com/guide/components/fundamentals.html
13. developer.android.com, (2013c), The AndroidManifest.xml File. Dostupno 25.8.2013. na http://developer.android.com/guide/topics/manifest/manifest-intro.html
14. developer.android.com, (2013d), Android emulator. Dostupno 25.8.2013. na http://developer.android.com/guide/topics/manifest/manifest-intro.html
15.
16. satworks.blogspot.com, (2013), Understanding Android Architecture & Project Structure. Dostupno 28.8.2013. na http://satworks.blogspot.com/2010/08/android-2-understanding-android.html

17. Dong Ho Han, (2013), Android, at a glance. Dostupno 28.8.2013. na http://www.cubrid.org/blog/dev-platform/android-at-a-glance/
18. Bornstain D, (2013), Dalmik VM Internals. Dostupno 28.8.2013. na https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf?attredirects=0
19. developer.nokia.com (2013), Porting Android (Java) applications to Java ME on S60 5th Edition. Dostupno 28.8.2013. na http://developer.nokia.com/Community/Wiki/Porting_Android_(Java)_applications_to_Java_ME_on_S60_5th_Edition
20. Dong Ho Han, (2013), Android, at a glance. Dostupno 16.6.2013. na http://www.cubrid.org/blog/dev-platform/android-at-a-glance/
21. stackoverflow.com, (2013), Point in Polygon aka hit test. Dostupno 16.6.2013. na http://stackoverflow.com/questions/217578/point-in-polygon-aka-hit-test/2922778#2922778
22. ormlite.com, (2013), OrmLite. Dostupno 16.6.2013 na http://ormlite.com/
23. kenfolios.com, (2013), Are you using the touch gestures of your touchscreen smartphone completely? Dostupno 2.8.2013 na http://www.kenfolios.com/mobiles-gadgets/touch-gestures/4566/
24. tomcat.apache.org , (2005), Apache Tomcat. Dostupno 19.7.2013 na http://tomcat.apache.org/
25. wiki.jenkins-ci.org, (2005), Meet Jenkins. Dostupno 19.7.2013 na https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins
26. searchcloudcomputing.techtarget.com, (2010), Platform as a Service (Paas). Dostupn 21.7.2013 na http://searchcloudcomputing.techtarget.com/definition/Platform-as-a-Service-PaaS
27. developer.android.com, (2013d), Displaying Graphics with OpenGL ES. Dostupno 8.9.2013 na http://developer.android.com/training/graphics/opengl/index.html
28. Google Inc., (2013), OpenGL ES examples. Dostupno 8.9.2013 na https://code.google.com/p/opengles-book-samples/