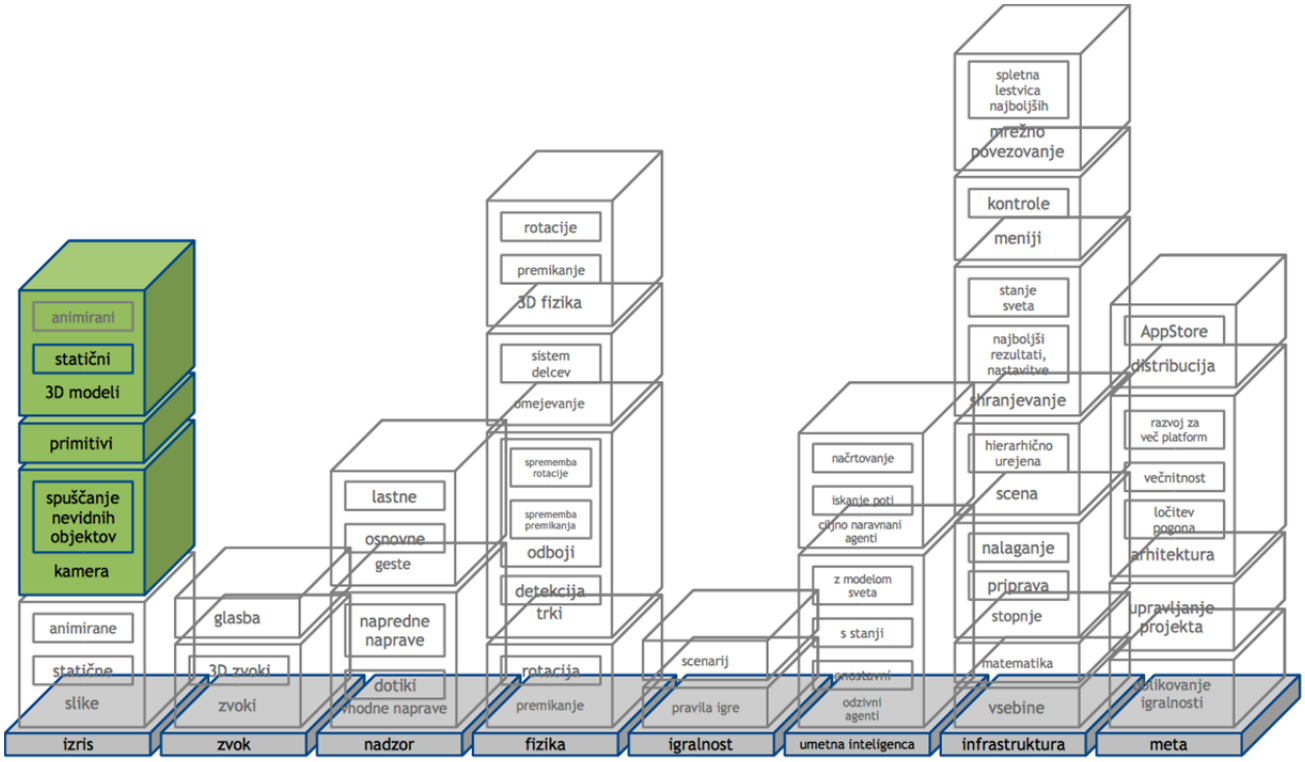
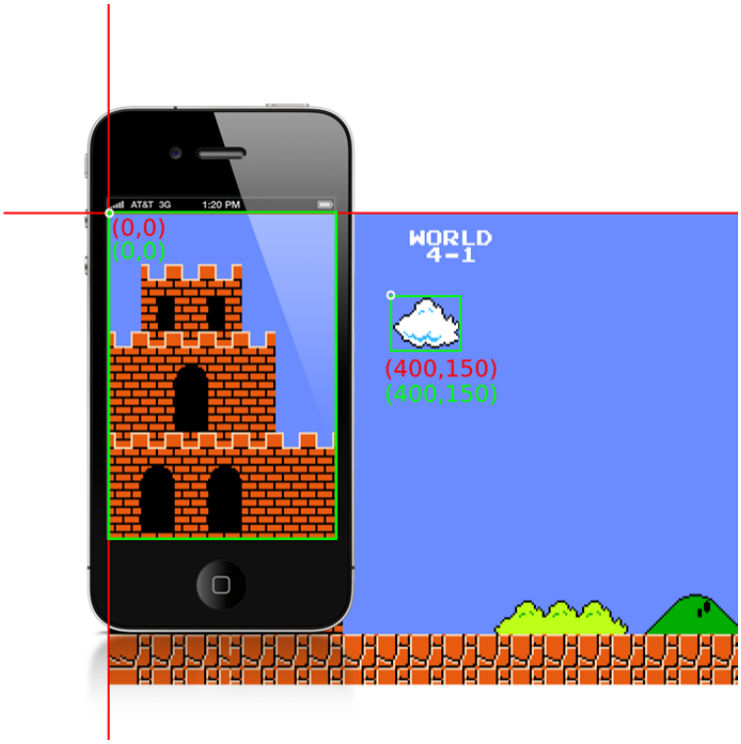
Sklop 7


animirani
statični
3D modeli
primitivi
spuščanje nevidnih objektov
kamera
animirane
statične
slike
izris
glasba
3D zvoki
zvoki
zvok
lastne
osnovne geste
napredne naprave
dotiki
vhodne naprave
nadzor
rotacije
premikanje
3D fizika
sistem delcev
omejevanje
sprememba rotacije
sprememba premikanja
odboji
detekcija trki
rotacija
premikanje
fizika
scenarij
pravila igre
igralnost
načrtovanje
iskanje poti
cilno naravnani agenti
z modelom sveta
s stanji
odzivni agenti
umetna inteligenca
spletna lestvica najboljših
mrežno povezovanje
kontrole
meniji
stanje sveta
najboljši rezultati, nastavitve
shranjevanje
hierarhično urejena
scena
nalaganje
priprava
stopnje
matematika
vsebine
infrastruktura
AppStore
distribucija
razvoj za več platform
večnitnost
ločitev pogona
arhitektura
upravljanje projekta
oblikovanje igralnosti
meta

Kamera

2D kamera

Ker zgolj v redkih igrah gledamo sceno le iz enega položaja je ključnega pomena vpeljava kamere. Lahko bi rekli, da smo že do sedaj imeli kamero z izhodiščem v zgornjem levem kotu (točki (0,0)) z 1-kratno povečavo, pri čemer nismo imeli možnosti spreminjanja njenih nastavitev. Koordinatni sistem sveta (rdeče črte in številke) se popolnoma sklada s koordinatnim sistemom ekrana (zelene črte in številke). Če oblak, ki se v svetu nahaja na koordinatah (400,150) hočemo izrisati na ekran, ga izrišemo na točno te koordinate (pojavil se bo zunaj ekrana, saj je ta širok le 320 pikslov):



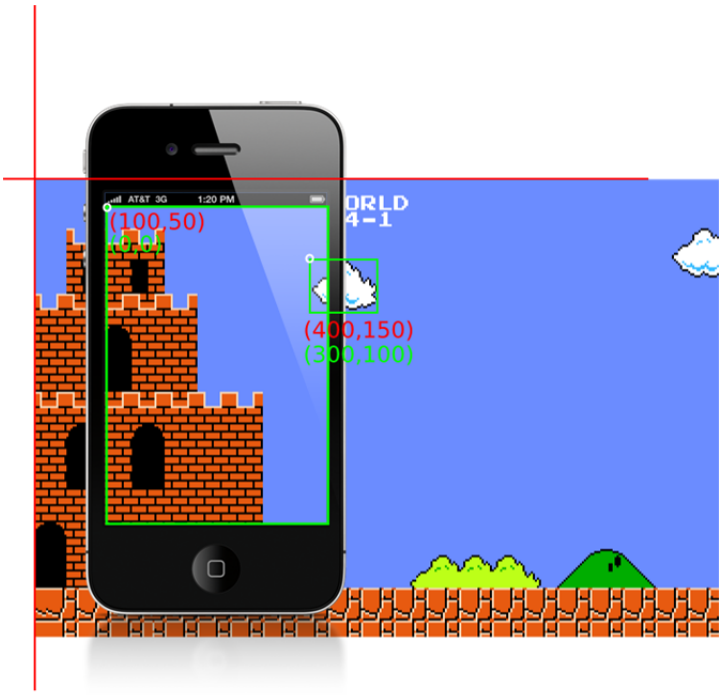
Da bomo lahko premaknili kamero, moramo najprej poskrbeti za spremenljivke, ki bodo opisovale ciljno točko kamere in povečavo.

```
Vector2 *cameraTarget;
float zoom;
```

Če zdaj kamero premaknemo za 100 pikslov v desno in 50 pikslov dol

```
cameraTarget = [Vector2 vectorWithX:150 y:50];
```

dobimo naslednjo sliko:

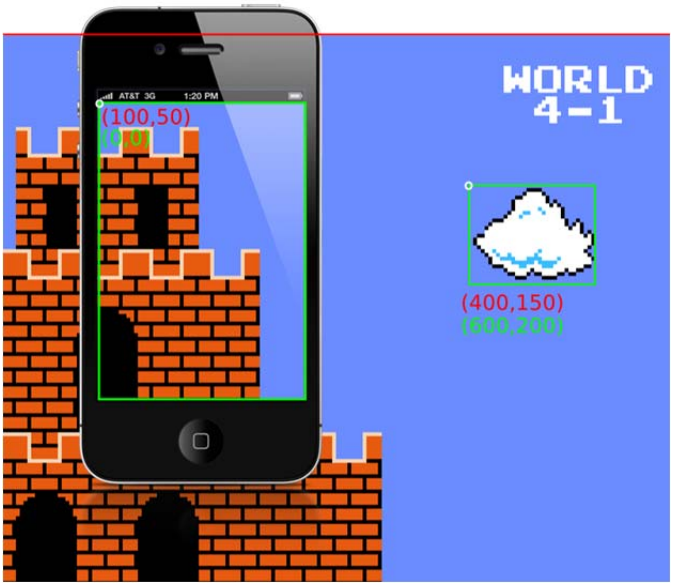


Oblak se je, relativno na koordinatni sistem ekrana, v katerem tudi sami izrisujemo z ukazi objekta SpriteBatch, premaknil za 100 pikslov v levo in 50 pikslov gor! Njegova pozicija v svetu (400,150) je pri kameri (100,50) spremenjena v pozicijo na ekranu (300, 100). Račun je seveda očiten: pri izrisu moramo od pozicije v svetu moramo vektor kamere odšteti:

```
Vector2 *cloudPosition = [Vector2 vectorWithX:400 y:150];

Vector2 *renderPosition = [Vector2 subtract:cloudPosition by:cameraTarget];
[spriteBatch draw:cloudTexture to:renderPosition tintWithColor:[Color white]];
```

Vpeljemo še povečavo, kar pomeni, da po izračunu razlike med pozicijo objekta in kamero to razdaljo še pomnožimo s faktorjem povečave. Tako bodo stvari blizu kamere še vedno ostale vidne, medtem ko bodo tiste na robu s povečevanjem zooma vedno bolj oddaljene in posledično padle izven meja ekrana. Če nastavimo vrednost zoom na 2, moramo pri isti kameri (100, 50) dobiti tako sliko:



Pozicija oblaka relativno na kamero je v svetu (300, 100), a pri izrisu moramo to pomnožiti še s povečavo, tako da dobimo vrednost (600, 200), kar zopet pade izven ekrana kot kaže zgornja slika. Spremenljivko zoom moramo uporabiti tudi pri samem izrisu, da bo oblak dejansko izrisan 2-krat večje:

```
Vector2 *cloudPosition = [Vector2 vectorWithX:400 y:150];

Vector2 *renderPosition = [Vector2 subtract:cloudPosition by:cameraTarget];
[renderPosition multiplyBy:zoom];

[spriteBatch draw:cloudTexture to:renderPosition fromRectangle:nil tintWithColor:[Color white] rotation:0 origin:nil
scaleUniform:zoom effects:SpriteEffectsNone layerDepth:0];
```

Podobno bi lahko dodali izračun za rotacijo (kotne funkcije), a s kombiniranjem pozicij, rotacij in povečav samih objektov z istimi operacijami na kameri, bi kmalu zašli v nepregledno solato kode. Ljubitelji linearne algebre boste tu že videli rešitev. Spremembo pozicije, povečavo in rotacijo lahko opišemo tudi z matrikami. Če te med sabo zmnožimo lahko z eno samo matriko opišemo spremembo vseh zaporednih operacij.

Zgornjo kodo premika kamere in povečave bi lahko prepisali takole:

```
Matrix *cameraTranslation = [Matrix createTranslationX:-cameraTarget.x y:-cameraTarget.y z:0];
Matrix *cameraZoom = [Matrix createScaleUniform:zoom];

Matrix *cameraTransform = [Matrix multiply:cameraTranslation by:cameraZoom];
```

S tako pripravljeno matriko kamere bi lahko zdaj vsako pozicijo spremenili s transformacijo:

```
Vector2 *renderPosition = [Vector2 transform:cloudPosition with:cameraTransform];
```

A stvari so z uporabo linearne algebre še boljše. *SpriteBatch* namreč omogoča, da vse operacije izrisa avtomatično transformira s podano matriko. Seveda je ta funkcionalnost dodana ravno za enostavno izdelavo kamere. Uporabimo jo preprosto tako, da ob začetku izrisa metodi *begin* podamo željeno matriko:

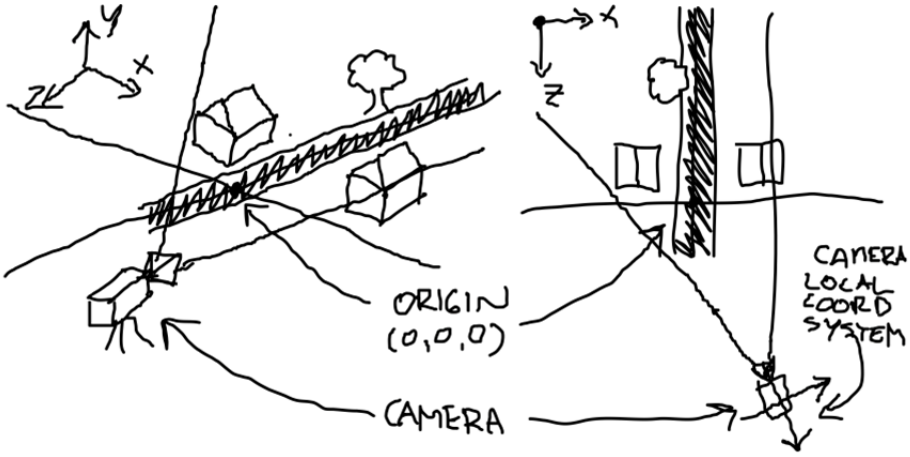
```
[spriteBatch beginWithSortMode:SpriteSortModeDeffered
                     BlendState:nil
                   SamplerState:nil
                DepthStencilState:nil
                  RasterizerState:nil
                           Effect:nil
                 TransformMatrix:cameraTransform];
```

3D kamera

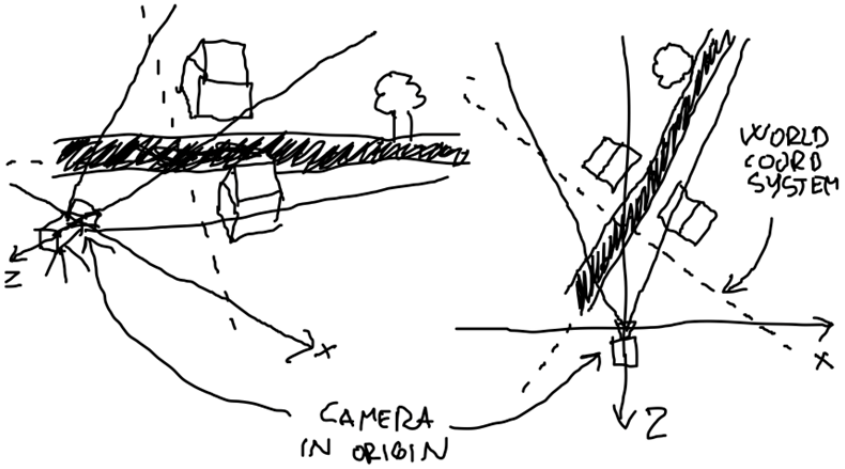
Čeprav se bomo s 3D grafiko začeli ukvarjati šele v naslednjem poglavju, je postavitev kamere prvi korak, preden sploh lahko kaj izrišemo. To je zelo pomembno poglavje, saj je osnova za razumevanje celotnega ukvarjanja s 3D grafiko.

S preskokom v tri dimenzije se nam predstavi nov problem, saj moramo 3D prostor prikazati na 2D ekranu, kar ni trivialno početje. Postopku rečemo projekcija, saj točke iz navideznega 3D prostora projiciramo na 2D zaslon, na pomoč pa nam bodo spet priskočile matrike, ki odlično opisujejo tudi projekcije.

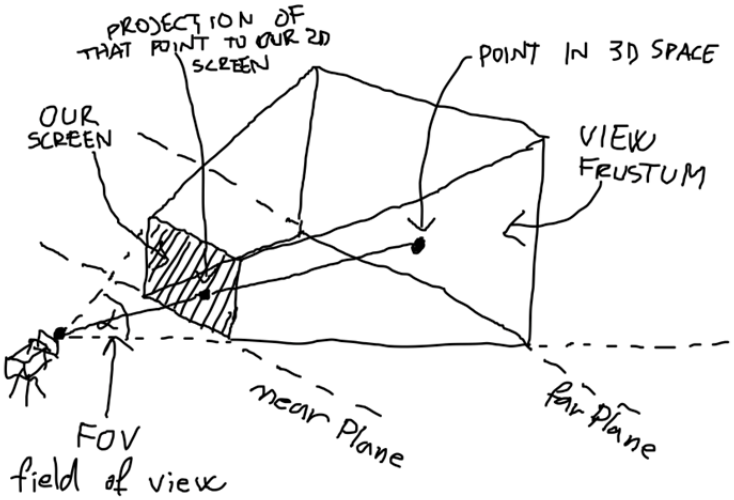
Najprej moramo vedeti da za vse pozicije objektov in postavitev kamere namesto *Vector2* uporabljamo objekte *Vector3*, ki imajo poleg X in Y še koordinato Z. V nadaljnjih ilustracijah bomo sceno predstavili tako v izometričnem pogledu (levi del ilustracije) kot tudi tlorisu (desni del ilustracije). Predstavljajte si torej cesto, dve hiši in drevo, ki jih snemamo pod kotom s kamero:



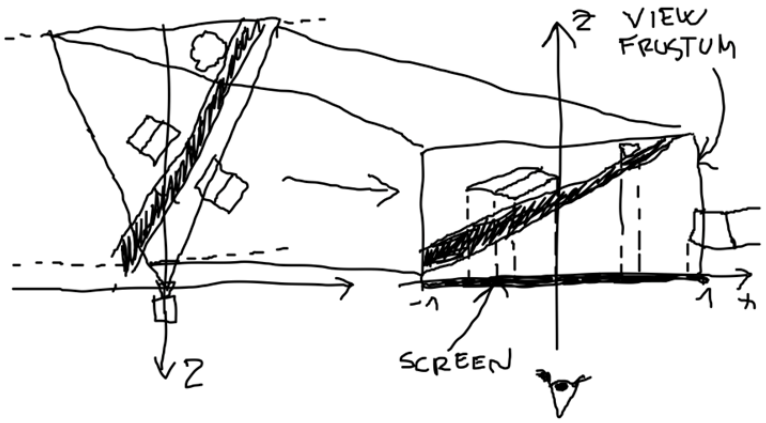
Transformacija iz poglavja o 2D kameri se običajno imenuje view transform, saj predstavlja transformacijo sveta glede na pogled (položaj kamere). Kot smo videli, iz koordinatnega sistema sveta (angl. world space) preračuna pozicije relativno na položaj kamere v koordinatni sistem kamere (angl. view space). Po taki transformaciji, bi naš svet izgledal takole:



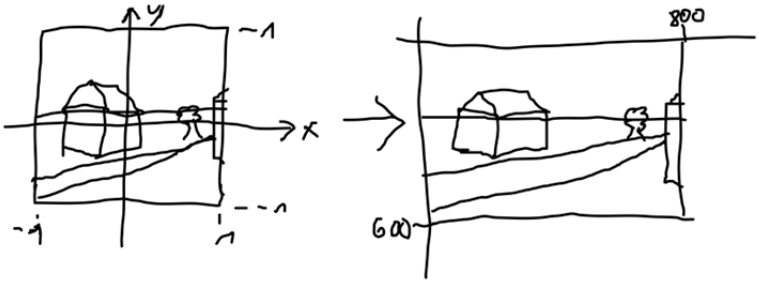
Nič presenetljivo drugače, le zarotiran in premaknjen je tako, da je kamera v izhodišču koordinatnega sistema in gleda vzdolž Z osi. Preden nadaljujemo je treba kameri določiti še tri stvari. Na sliki je že označen kót širine pogleda, ki pove, kako širok del scene vidimo. Nadalje je treba dobljeno piramido še omejiti od kje do kje bo vidni del scene, kar naredimo z dvema ravninama – bližnjo in oddaljeno (near in far plane). Dobimo piramido z odsekanim vrhom (frustum):



Če si zdaj predstavljamo, da je pravokotnik na sprednji ravnini kar površina našega ekrana, moramo vse točke znotraj piramide projicirati na bližnjo ravnino. To stori projekcijska transformacija (projection). V našem prejšnjem primeru bi to zgledalo takole:



Vidimo da se vidni del sveta spremeni iz odsekane piramide v kvader, kar ustrezno popači razdalje. Stvari, ki so bile bliže ekranu so kar naenkrat večje (hiši), predmeti v oddaljenosti pa veliko manjši (drevo). To je točno to kar želimo – perspektiva. Če koordinatam v koordinatnem sistemu projekcije (angl. projection space) zanemarimo Z koordinato in glede na X in Y narišemo sliko v dveh dimenzijah smo dobili našo 3D sliko sploščeno na 2D ekran:



Odlična stvar vsega skupaj je, da nam ni potrebno ročno ustvarjati matrik *view* in *projection*, saj so rutine za njihovo izdelavo že na voljo preko razreda Matrix, pri čemer mi samo podamo glavne podatke (pozicijo kamere, ciljno točko pogleda, kót pogleda in sprednjo ter zadnjo ravnino):

```
Matrix *projection = [Matrix createPerspectiveFieldOfView:M_PI * 0.13f
                                                  aspectRatio:self.graphicsDevice.viewport.aspectRatio
                                              nearPlaneDistance:1
                                               farPlaneDistance:100];

Matrix *view = [Matrix createLookAtFrom:cameraPosition to:cameraTarget up:[Vector3 up]];
```

Naloga: kamera

Pogosto je nemogoče spraviti celotno igralno področje na velikost zaslona. Tako vidimo le del sveta naenkrat. Kateri je to, določamo s pomočjo navidezne kamere. Dodatno lahko kamero izrabimo tudi pri enozaslonskih igrah, recimo za učinek tresenja ekrana ob eksploziji.

V 2D svetu je najosnovnejša lastnost kamere točka v katero imamo usmerjen pogled. Iz tega vektorja s pomočjo metode *[Matrix createTranslation:]* ustvarimo matriko, ki vse koordinate v svetu pretvori tako, da se bo ciljna točka sveta pojavila v vidnem polju ekrana. Matriko uporabimo pri klicu *[SpriteBatch begin: ... transformMatrix:]*.

Dodatno lahko v 2D svetu s pomočjo *[Matrix createScale:]* dosežemo približevanje in oddaljevanje pogleda, z *[Matrix createRotationZ:]* pa vrtenje celotne slike. Seveda lahko uporabimo vse efekte skupaj, če tri delne matrike (premik, vrtenje, povečava) zmnožimo v pravilnem vrstnem redu.

Kamera v 3D svetu deluje drugače in je opisana z dvema posebnima matrikama, pogledom (*view*) in projekcijo (*projection*). Matriko pogleda običajno ustvarimo s klicem *[Matrix createLookAtFrom: to:*

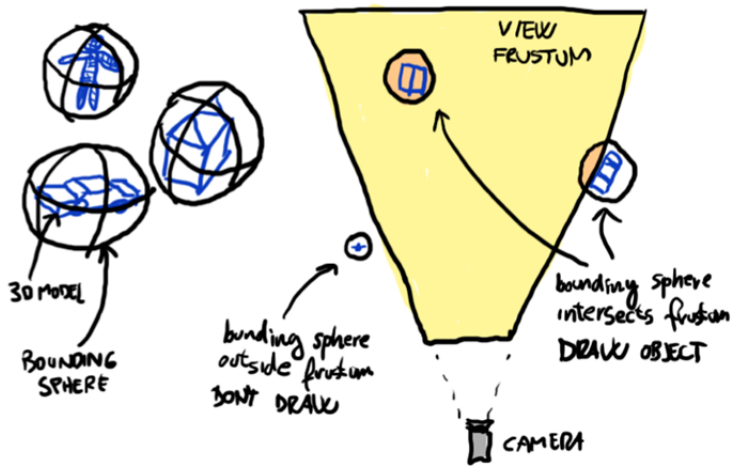
up:], projekcijo pa z *[Matrix createPerspectiveFieldOfView: aspectRatio: nearPlaneDistance: farPlaneDistance]*.

Če za kamero ustvarimo svoj razred, bomo običajno na njej nastavljali lastnosti kot so pozicija kamere, točka, ki jo snemamo, in kót pogleda. Na podlagi teh podatkov nato ustrezno izračunamo matriki view in projection.

Kakorkoli pristopimo k stvari, končni rezultat izdelave navidezne kamere je, da se pogled na svet v igri lahko spreminja.

Spuščanje nevidnih objektov

Ko začnemo izrisovati svetove večje od zaslona, se moramo vprašati, ali ne bi lahko pospešili izrisovanja, če objektov zunaj vidnega polja kamere preprosto ne izrišemo. "Frustum culling" ali po domače, ne riši, česar ne vidiš, je matematični izračun, s katerim pred klicanjem izrisa na grafični kartici vnaprej ugotovimo, ali bo objekt po projekciji padel na ekran ali ne. Veliko ceneje, kot transformirati vsako oglišče vseh trikotnikov je, če 3D modelu orišemo kroglo (angl. bounding sphere) in izračunamo samo, če krogla seka vidni stožec (angl. frustum) ali ne.



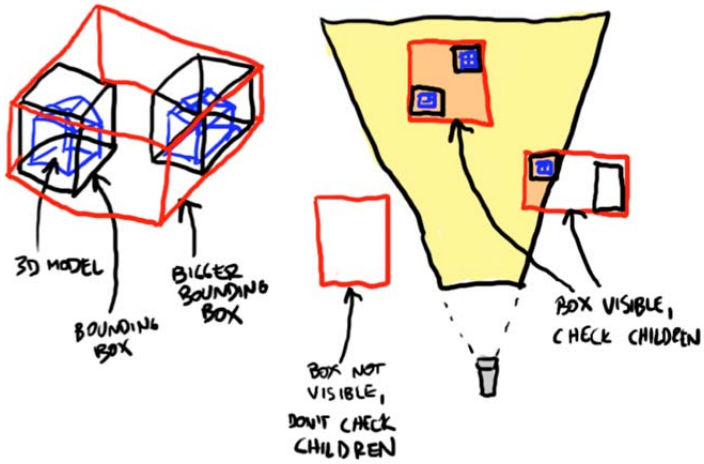
Ker orisana krogla predstavlja nekoliko večjo prostornino, kot jo zavzema sam 3D model, se včasih seveda zgodi, da je krogla vidna, objekt, katerega orisuje, pa ne. V tem primeru sicer še vedno rišemo nekaj, kar se ne vidi, vendar se to zgodi za zelo majhno število objektov. Bolj pomembno je, da krogla res vsebuje celoten model, saj bi v nasprotnem primeru lahko bil kakšen del modela viden, a se ne bi izrisal, ker bi bila njegova predstavitev s kroglo že izven ekrana.

Enako tehniko lahko uporabimo tudi v 2D svetu, kjer je naš pogled kamere pravokotnik, like pa namesto s kroglami orišemo s pravokotniki.

Deli in vladaj v praksi (napredno spuščanje nevidnih objektov)

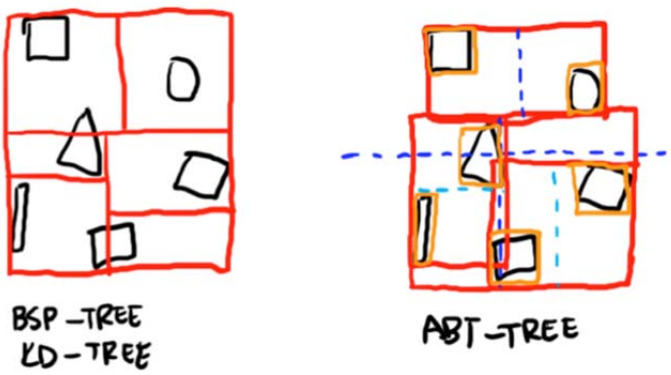
Izris smo že optimizirali z uporabo izpuščanja nevidnih objektov, vendar to ne bi bilo dovolj, če bi na sceni imeli tisoč in več objektov, saj bi računanje sekanja krogle z vidnim stožcem celokupno še vedno trajalo preveč časa, za vseh tisoč poizvedb. Zato si bomo sposodili dobro poznan algoritem deli in vladaj ter sceno hierarhično predstavili z dvojiškim drevesom.

Predstavljajte si, da za začetek namesto s kroglo vsak predmet orišemo s kvadrom, torej namesto *BoundingSphere* uporabljamo *BoundingBox*. Za vsak objekt v svetu preverimo, ali je njegov kvader znotraj vidnega stožca, in če je, ga izrišemo, sicer ne. Sedaj vpeljemo izboljšavo. Po dva bližnja objekta nadomestimo z večjo škatlo, ki ravno vključuje oba manjša kvadra. Zdaj najprej preverimo za večji BoundingBox ali je znotraj vidnega polja, in če ni, nam ni potrebno posebej preverjati za obe manjši škatli znotraj. V nasprotnem primeru pa preverimo še vsako posebej.



Zdaj na ta način večje škatle združimo v še večje in to ponavljamo dokler ne pridemo do ene same škatle. Ali, če se stvari raje lotimo z obratnega stališča, začnemo z našim svetom v katerem je 200 objektov in jih razdelimo na dve polovici. Obe polovici še nadalje razdelimo in to ponavljamo dokler nimamo v vozlišču dovolj majhno število objektov.

Tako smo dobili dvojiško drevo (angl. binary tree) s seznamom objektov v listih. Če hočemo izrisati sceno, zgolj rekurzivno po principu deli in vladaj pogledamo, ali sta vidni leva in desna polovica, in za tisto ki je, sprožimo pregledovanje sinov. Ko pridemo do listov izrišemo vse objekte, ki se nahajajo v listu. Problem je, da smo pri delitvi zvrha navzdol sceno vsakič razdelili z neko ravnino na dve polovici. Prej ali slej se pojavi kakšen objekt, ki leži ravno na sredini in torej spada tako v levo, kot desno polovico. Da nam ni treba objekta podvajati v obeh straneh, ustvarimo prilagodljivo dvojiško drevo, znano pod kratico ABT (angl. adaptive binary tree). Pri tem pristopu objekt postavimo v samo eno od polovic in njenega orisnega kvadra povečamo toliko, da vsebuje tudi nov objekt. Seveda se orisana kvadra obeh polovic pri tem prekrivata, vendar s tem ni nič narobe. Le vsakič, ko v sceno dodamo nov objekt moramo orisani kvader prilagoditi, če objekt slučajno sega čez rob.



Pridobitev hitrosti pri izrisu je občutna. Naš prvotni pristop s preverjanjem obsegajoče krogle za vsak objekt posebej ima časovno zahtevnost reda O(n), pri čemer je n število objektov v sceni. Z uporabo dvojiškega drevesa in algoritma deli in vladaj, naredimo iskanje objektov v našem vidnem polju velikostnega reda O(log n).

Če bi 400 objektov povečali na 10.000 objektov na sceni, bi pri O(n) pristopu potrebovali 10.000/400=25 krat več časa. Pri redu O(log n) bi enako povečanje potrebovalo samo log(10.000)/log(400)=1,5-krat več časa.

V praksi se izkaže, da če namesto z 200 objekti ustvarimo svet s 50.000 objekti, brez ABT-ja FPS pade na okoli 6 sličic na sekundo, z ABT-jem pa je FPS še vedno višji od 200. Vsekakor dovolj, da smo zelo praktično prikazali uporabnost snovi, ki se mogoče marsikateremu študentu računalništva zdi brez večje uporabne vrednosti v praksi.

Naloga: spuščanje nevidnih objektov

V igrah, kjer ne vidimo celotnega sveta naenkrat, lahko pospešimo izris, če za objekte, ki se nahajajo izven vidnega polja, sploh ne pokličemo kode za izris. GPU šele po transformaciji oglišč ugotovi, da posamezen trikotnik ne bo viden. Zato lahko prihranimo delo, če matematiko prenesemo na CPU in za celoten objekt naenkrat izračunamo, ali je v vidnem polju. To je veliko hitreje, saj objekt pred tem poenostavimo z orisanjem krogle ali kvadra, oziroma kroga ali pravokotnika v dveh dimenzijah. Če ta geometrični primitiv primerjamo z vidnim poljem (vidni stožec – frustum v 3D, pravokotnik v 2D), bomo z izračunom geometričnega preseka ugotovili, ali je objekt potencialno viden.

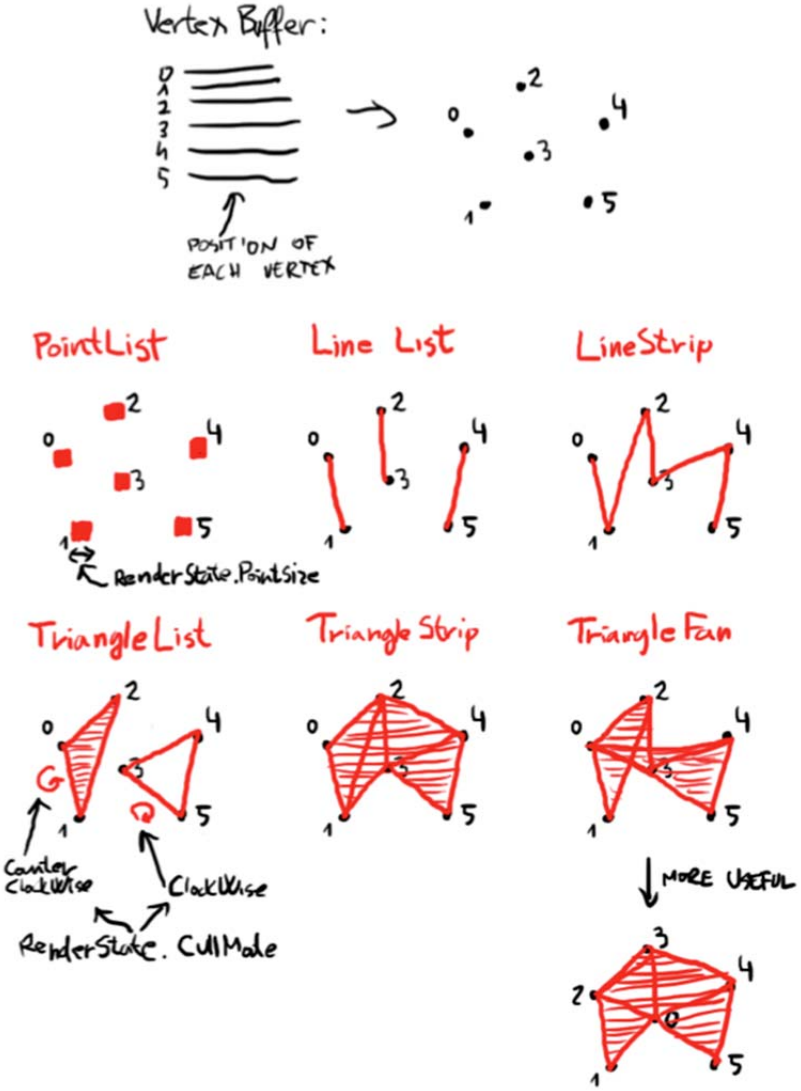
Pri ugotavljanju točk sveta, ki ustrezajo ogliščem ekrana, si pomagamo s klicem *[graphicsDevice.viewport unproject: projection: view: world:]*, da iz koordinat zaslona pridemo v koordinatni sistem sveta.

Grafični primitivi

Zdaj se zares začnemo spuščati v 3D grafiko, ki je danes tudi pod pokrovom 2D izrisa. Kako *SpriteBatch* nariše sliko na zaslon? Od grafičnega čipa zahteva, naj nariše dva trikotnika, ki jih združi v pravokotnik. Osnovnim likom, ki jih grafični čip zna izrisati, rečemo grafični primitivi. To so točke, črte in trikotniki, čeprav je XNA podporo za točke že prenehal, saj različni proizvajalci zelo različno podpirajo to sposobnost. Za črte in trikotnike obstaja še več načinov, v kakšnem vrstnem redu podamo oglišča za opis likov, kakor tudi načinov za podajanje teh podatkov.

Vrste grafičnih primitivov

V vseh različicah izrisa grafični kartici podamo seznam oglišč in tip primitivov, ki jih ta oglišča opisujejo. Poglejmo si kar vizualno, kako različni tipi interpretirajo oglišča:



Če bi radi izrisali poljubne črte, izberemo tip *LineList* in v seznam oglišč zaporedno dodamo začetno in končno koordinato črte. Podobno deluje *TriangleList* za trikotnike - za vsakega podamo tri zaporedna oglišča, pri čemer moramo paziti še na orientiranost (v smeri urinega kazalca ali v nasprotni smeri), od česar je odvisno, s katere strani bo viden trikotnik. To lastnost lahko spremenimo z uporabo drugih nastavitev rasterizatorja:

```
self.graphicsDevice.rasterizerState = [RasterizerState cullNone];
self.graphicsDevice.rasterizerState = [RasterizerState cullClockwise];
self.graphicsDevice.rasterizerState = [RasterizerState cullCounterClockwise];
```

Če želimo izrisati povezano črto, lahko uporabimo grafični primitiv *LineStrip*. Podobno deluje TriangleStrip, s tem da ob vsakem novem oglišču uporabi še prejšnja dva za sestavo trikotnika. Dobimo torej trikotnike z oglišči na mestih (0, 1, 2), (1, 2, 3), (2, 3, 4) ... (n-2, n-1, n).

TriangleFan primitiv tako kot *PointList* XNA ne podpira več, zato ju tudi XNI ne podpira, vendar pa jih še lahko uporabite, če za tip podate direktno OpenGL konstanti *GL_POINTS* in *GL_TRIANGLE_FAN*.

Vrste oglišč

Preden lahko skočimo na postopek za izris moramo vedeti še kakšne podatke hrani vsako oglišče. Izbir je več, odvisno od tega, ali bomo uporabljali osvetljavo ali vnaprej pobarvana oglišča, ter če potrebujemo teksturirane trikotnike ali ne. Vsako oglišče ima lahko nekatere od teh podatkov:

- position: 3D koordinata, ki jo zahtevajo vsa oglišča.
- color: vnaprej podana barva, ki naj jo ima oglišče.
- normal: normala površine, s katero se izračuna barva glede na osvetljevanje.
- texture: 2D teksturna koordinata, ki označuje, kje na teksturi je to oglišče (med 0 in 1).

XNI razpolaga z običajnimi kombinacijami in za vsako pripravlja c-jevsko strukturo (struct) za samo hranjenje podatkov, razred, ki nosi še dodatne meta podatke ter seznam, v katerega shranjujemo zaporedna oglišča:

struktura	**razred**	**seznam**
VertexPositionColorStruct	VertexPositionColor	VertexPositionColorArray
VertexPositionTextureStruct	VertexPositionTexture	VertexPositionTextureArray
VertexPositionColorTextureStruct	VertexPositionColorTexture	VertexPositionColorTextureArray
VertexPositionNormalTextureStruct	VertexPositionNormalTexture	VertexPositionNormalTextureArray

Priprava seznama podatkov

Za začetek vzemimo primer izrisa pobarvane črte (brez senčenja). Najprej si pripravimo seznam za shranjevanje oglišč:

```
VertexPositionColorArray *vertexArray = [[VertexPositionColorArray alloc] initWithInitialCapacity:5];
```

V strukturo damo podatke prvega oglišča in ga dodamo v seznam:

```
VertexPositionColorStruct colorVertex;

colorVertex.position.x = -1;
colorVertex.position.y = 0;
colorVertex.position.z = -1;
colorVertex.color = [Color blue].packedValue;
[vertexArray addVertex:&colorVertex];
```

Spremenimo podatke in znova dodamo oglišče (podatki se v seznam kopirajo), dokler nismo napolnili vseh želenih podatkov:

```
colorVertex.position.x = -1;
colorVertex.position.y = 0;
colorVertex.position.z = 1;
colorVertex.color = [Color yellow].packedValue;
[vertexArray addVertex:&colorVertex];

colorVertex.position.x = 1;
colorVertex.position.y = 0;
colorVertex.position.z = 1;
colorVertex.color = [Color red].packedValue;
[vertexArray addVertex:&colorVertex];
```

```
colorVertex.position.x = 1;
colorVertex.position.y = 0;
colorVertex.position.z = -1;
colorVertex.color = [Color lime].packedValue;
[vertexArray addVertex:&colorVertex];

colorVertex.position.x = -1;
colorVertex.position.y = 0;
colorVertex.position.z = -1;
colorVertex.color = [Color blue].packedValue;
[vertexArray addVertex:&colorVertex];
```

Senčilnik brez osvetlitve

Preden lahko oglišča izrišemo, moramo pripraviti senčilnik, ki bo povedal, na kakšen način naj se izrišejo primitivi (nastavitev kamere, uporaba tekstur, uporaba senčenja ...).

Uporabili bomo razred BasicEffect in mu nastavili lastnosti za izris že pobarvanih (vertex colors) primitivov:

```
BasicEffect *colorEffect = [[BasicEffect alloc] initWithGraphicsDevice:self.graphicsDevice];
colorEffect.world = [Matrix createTranslationX:-3 y:0 z:0];
colorEffect.view = view;
colorEffect.projection = projection;
colorEffect.vertexColorEnabled = YES;
```

Tukaj torej uporabimo matriki *view* in *projection*, ki smo jih pripravili v poglavju o 3D kameri. Matrika world nam dodatno predstavlja, kam v svet želimo postaviti naša oglišča. Če uporabimo identiteto, se bodo oglišča, kot smo jih shranili v seznam točno ujemala z globalnim koordinatnim sistemom. Z uporabo matrike lahko 3D model premaknemo na drugo pozicijo/rotacijo/povečamo-zmanjšamo.

Izris

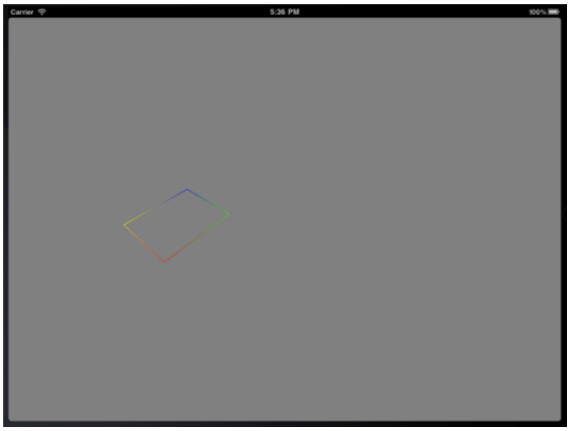
S pripravljenimi podatki in senčilnikom smo pripravljeni, da knjižnici ukažemo izris. Pred izrisom moramo ukazati senčilniku naj vzpostavi svoje nastavitve, pred tem pa običajno še nastavimo nastavitev matrike world za konkretni objekt v svetu, ki ga želimo izrisati. Recimo, da želimo, da se naše črte vrtijo, bomo uporabili sledeč postopek za izris:

```
colorEffect.world = [Matrix multiply:[Matrix createRotationY:gameTime.totalGameTime]
                                                          by:[Matrix createTranslationX:-3 y:0 z:0]];

[[colorEffect.currentTechnique.passes objectAtIndex:0] apply];

[self.graphicsDevice drawUserPrimitivesOfType:PrimitiveTypeLineStrip
                                   vertexData:vertexArray
                                 vertexOffset:0
                                primitiveCount:4];
```

Dobimo vrteč pravokotnik iz pobarvanih črt, ki se vrti okoli točke (-3, 0, 0):



Indeksirani podatki

Če bi poskusili izrisati 3D kocko iz črt, bi porabili 12 primitivov in pri načinu *LineList* 24 oglišč. Kocka ima seveda samo 8 različnih oglišč, tako da vsako oglišče pošiljamo trikrat.

Podoben primer je pri izrisu pravokotnika iz dveh trikotnikov. Imamo 4 oglišča, če jih pošljemo v običajnem načinu *TriangleList* moramo podatke o dveh ogliščih poslati dvakrat. Kompleksnejše 3D modele kot imamo, več podvajanja se pojavlja. Da ni potrebno identičnih podatkov pošiljati večkrat, lahko uporabimo indeksiranje oglišč.

Pri tem načinu vsako oglišče zapišemo samo enkrat v naš seznam in si za vsako oglišče zapomnimo indeks – mesto, na katerega smo v seznam shranili oglišče. Tokrat bomo prikazali izris teksturiranega kvadrata s senčenjem:

```
VertexPositionNormalTextureArray *texturedVertexArray = [[VertexPositionNormalTextureArray alloc]
initWithInitialCapacity:4];
VertexPositionNormalTextureStruct vertex;

vertex.position.x = -1;
vertex.position.y = 0;
vertex.position.z = -1;
vertex.normal.x = 0;
vertex.normal.y = 1;
vertex.normal.z = 0;
vertex.texture.x = 0;
vertex.texture.y = 0;
[texturedVertexArray addVertex:&vertex];

vertex.position.x = -1;
vertex.position.y = 0;
vertex.position.z = 1;
vertex.normal.x = 0;
vertex.normal.y = 1;
vertex.normal.z = 0;
vertex.texture.x = 0;
vertex.texture.y = 1;
[texturedVertexArray addVertex:&vertex];

vertex.position.x = 1;
vertex.position.y = 0;
vertex.position.z = 1;
vertex.normal.x = 0;
vertex.normal.y = 1;
vertex.normal.z = 0;
vertex.texture.x = 1;
vertex.texture.y = 1;
[texturedVertexArray addVertex:&vertex];

vertex.position.x = 1;
vertex.position.y = 0;
vertex.position.z = -1;
vertex.normal.x = 0;
vertex.normal.y = 1;
vertex.normal.z = 0;
vertex.texture.x = 1;
vertex.texture.y = 0;
[texturedVertexArray addVertex:&vertex];
```

Shranili smo si torej zgornje-levo (indeks 0), spodnje-levo (1), spodnje-desno (2) in zgornje-desno oglišče (3).

Sedaj uporabimo novo podatkovno strukturo IndexArray, v shranimo zgolj seznam indeksov oglišč. Da povežemo oglišča v dva trikotnika (0,2,1) in (2,0,3) sestavimo naslednji seznam indeksov:

```
ShortIndexArray *indexArray = [[ShortIndexArray alloc] initWithInitialCapacity:6];
[indexArray addIndex:0];
[indexArray addIndex:2];
[indexArray addIndex:1];
[indexArray addIndex:2];
[indexArray addIndex:0];
[indexArray addIndex:3];
```

Senčilnik z osvetlitvijo

Preden lahko izrišemo naš drugi primer, ki uporablja teksturne koordinate in osvetljevanje, moramo pripraviti nov senčilnik in mu nastaviti vir svetlobe:

```
BasicEffect *effect = [[BasicEffect alloc] initWithGraphicsDevice:self.graphicsDevice];
effect.world = [Matrix createTranslationX:3 y:0 z:0];
effect.view = view;
effect.projection = projection;
effect.textureEnabled = YES;
effect.vertexColorEnabled = NO;
effect.lightingEnabled = YES;

effect.texture = [self.content load:@"textureName"];
effect.diffuseColor.x = 1;
effect.diffuseColor.y = 1;
effect.diffuseColor.z = 1;

effect.ambientLightColor.x = 0.7;
effect.ambientLightColor.y = 0.6;
effect.ambientLightColor.z = 0.8;

effect.directionalLight0.enabled = YES;
effect.directionalLight0.direction = [Vector3 down];
effect.directionalLight0.diffuseColor.x = 1;
effect.directionalLight0.diffuseColor.y = 1;
effect.directionalLight0.diffuseColor.z = 1;
```

Nastavimo lahko do 3 vire svetlobe.

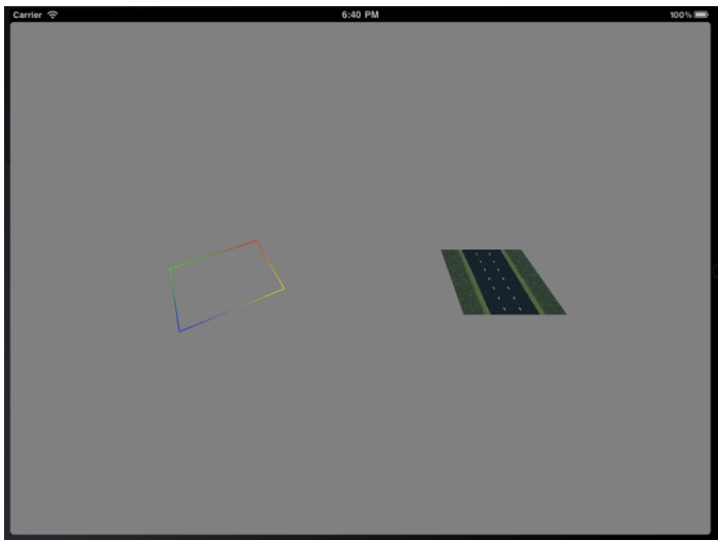
Izris indeksiranih grafičnih primitivov

Tokrat namesto *drawUserPrimitives* uporabimo ukaz *drawUserIndexedPrimitives* in namesto obračanja modela spreminjamo kót luči:

```
effect.directionalLight0.direction.y = sinf(gameTime.totalGameTime);
effect.directionalLight0.direction.z = cosf(gameTime.totalGameTime);
[[effect.currentTechnique.passes objectAtIndex:0] apply];

[self.graphicsDevice drawUserIndexedPrimitivesOfType:PrimitiveTypeTriangleList
                                          vertexData:texturedVertexArray
                                        vertexOffset:0
                                         numVertices:4
                                             indexData:indexArray
                                         indexOffset:0
                                       primitiveCount:2];
```

Poleg prejšnjega črtnega modela dobimo na desni teksturiran, osenčen pravokotnik:



Shranjevanje podatkov v grafičnem pomnilniku

Na način, kot smo zdaj prikazali izris, moramo za vsako sliko na novo poslati podatke o ogliščih (in indeksih). Če izrisujemo podatke, ki se vsako sliko tudi spreminjajo, je to običajno. Za statične podatke pa je veliko bolje, če jih samo enkrat pošljemo grafičnemu čipu, nakar se podatki tam shranijo.

Za shranjevanje v grafični pomnilnik imamo na voljo razreda *VertexBuffer* in *IndexBuffer*. Podatke iz seznamov precej enostavno shranimo v njih:

```
VertexBuffer *vertexBuffer = [[VertexBuffer alloc] initWithGraphicsDevice:self.graphicsDevice
                                                                    vertexDeclaration:texturedVertexArray.vertexDeclaration
                                                                          vertexCount:4
                                                                                usage:BufferUsageWriteOnly];
[vertexBuffer setData:texturedVertexArray];

// Index buffer
IndexBuffer *indexBuffer = [[IndexBuffer alloc] initWithGraphicsDevice:self.graphicsDevice
                                                             indexElementSize:IndexElementSizeSixteenBits
                                                                   indexCount:6
                                                                        usage:BufferUsageWriteOnly];
[indexBuffer setData:indexArray];
```

Zdaj ukaz drawUserPrimitives (in drawUserIndexedPrimitives) zamenjamo z drawPrimitives (in drawIndexedPrimitives). Pred tem moramo nastaviti katera dva medpomnilnika naj se uporabita za izris. Recimo za prejšnji primer:

```
[self.graphicsDevice setVertexBuffer:vertexBuffer];
self.graphicsDevice.indices = indexBuffer;

[self.graphicsDevice drawIndexedPrimitivesOfType:PrimitiveTypeTriangleList
                                      baseVertex:0
                                   minVertexIndex:0
                                     numVertices:4
                                      startIndex:0
                                   primitiveCount:2];
```

Vizualno je rezultat seveda enak, razlika je v performancah pri velikih modelih.

Naloga: primitivi

Včasih vsega ni mogoče izrisati zgolj s slikami, temveč moramo poseči po najosnovnejših grafičnih primitivih, katerih izris omogoča grafični čip. To so črte in trikotniki, s pomočjo katerih lahko izrisujemo spreminjajoče krivulje, posebne učinke (dež, iskre), večkotnike z dinamično barvo oglišč itd.

Za izris grafičnih primitivov si moramo pripraviti vsaj seznam oglišč, ki bodo sestavljali primitive, in senčilnik (*BasicEffect*) – z njim določimo transformiranje oglišč (preračunavanje iz 3D koordinat sveta na 2D zaslon preko matrik *world*, *view* in *projection*) in izračun barve posameznih pikslov (preko barve materiala in osvetlitve).

Izdelamo si lahko tudi splošno komponento, ki deluje podobno kot S*priteBatch*, le da namesto dodajanja trikotnikov za izris slik, pripravi potrebna oglišča za izris črt, pravokotnikov, krogov in podobnih pogosto uporabljanih likov. Primer tako izdelane komponente je *PrimitiveBatch* iz projekta Artificial I.

3D modeli

Izris grafičnih primitivov razlaga tehnično ozadje izrisovanja 3D modelov. V praksi, razen pri naprednejših tehnikah, kot je izris 3D terena, ne sestavljamo oglišč ročno iz kode. Uporabimo namreč zunanje 3D modele, ki jih ustvarijo oblikovalci v programih za modeliranje.

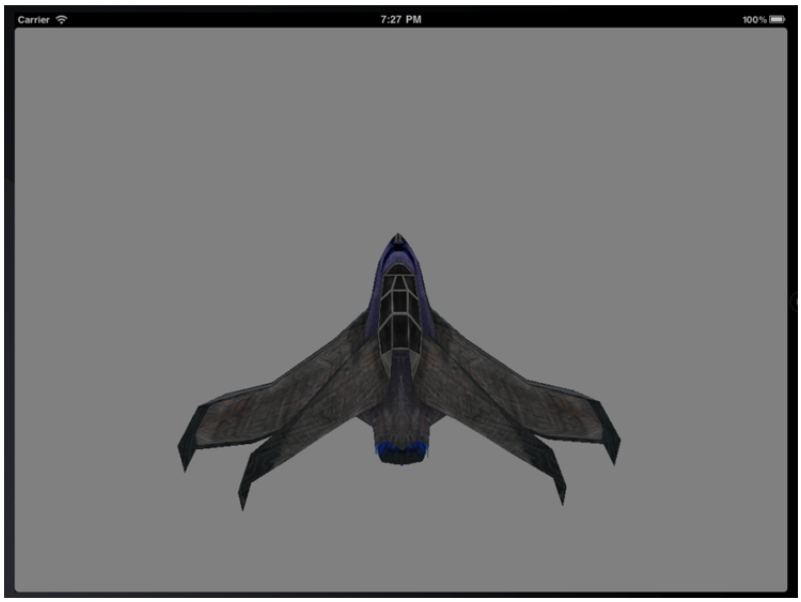
Nalaganje in izris je zelo preprosto, saj se samodejno v ozadju ustvarijo vsi medpomnilniki in senčilniki:

```
Model *model = [self.content load:@"defendermodel"];

Matrix *world = [Matrix multiply:[Matrix createRotationY:gameTime.totalGameTime]
                              by:[Matrix createRotationZ:gameTime.totalGameTime/2.0f]];

[model drawWithWorld:world view:view projection:projection];
```

Matriko *world* lahko seveda sestavimo kakor želimo. Končni rezultat je izrisan 3D model.



Če hočemo spremeniti osvetljavo, moramo na vseh senčilnikih v ozadju spremeniti podatke:

```
for (ModelMesh *mesh in model.meshes) {
    for (BasicEffect *effect in mesh.effects) {
        effect.vertexColorEnabled = NO;
        effect.lightingEnabled = YES;

        effect.ambientLightColor.x = 0.7;
        effect.ambientLightColor.y = 0.6;
        effect.ambientLightColor.z = 0.8;

        effect.directionalLight0.enabled = YES;
        effect.directionalLight0.direction = [Vector3 left];
        effect.directionalLight0.diffuseColor.x = 0;
        effect.directionalLight0.diffuseColor.y = 0;
        effect.directionalLight0.diffuseColor.z = 1;

        effect.directionalLight1.enabled = YES;
        effect.directionalLight1.direction = [Vector3 right];
        effect.directionalLight1.diffuseColor.x = 1;
        effect.directionalLight1.diffuseColor.y = 0;
        effect.directionalLight1.diffuseColor.z = 0;

        effect.directionalLight2.enabled = YES;
        effect.directionalLight2.direction = [Vector3 down];
        effect.directionalLight2.diffuseColor.x = 0.5;
        effect.directionalLight2.diffuseColor.y = 0.5;
        effect.directionalLight2.diffuseColor.z = 0.3;
    }
}
```

Tako dobimo osvetljen rezultat:

Naloga: statični 3D modeli

Izris 3D grafike je z XNI zelo enostaven. Vse kar potrebujemo je 3D model shranjen v tekstovnem .x formatu. S *ContentManager*jem ga naložimo v igro in tako dobimo objekt tipa *Model*. Z matrikami kamere (*view*, *projection*) in postavitvijo v svet (shranjeno v matriko *world*) tak 3D model izrišemo z ukazom *[model drawWithWorld: view: projection:]*.

Na ta način enako kot pri 2D izrisu pripravimo *Renderer*, le da namesto s slikami in *SpriteBatch*om objekte izriše s pripravljenimi 3D modeli.