Tomi Karppinen

XNA-ohjelmointi

Tietotekniikan
kandidaatintutkielma
11. lokakuuta 2010

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Tomi Karppinen
Yhteystiedot: tomi.j.karppinen@jyu.fi
Työn nimi: XNA-ohjelmointi
Title in English: Programming XNA
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 57
Tiivistelmä: XNA on Microsoftin vuonna 2006 julkaisema uudenlainen rajapinta helpompaan ja käyttäjäläheisempään peliohjelmointiin Windows-käyttöjärjestelmälle, Xbox 360 -konsolille ja Zune-mediasoittimelle. Tässä tutkielmassa käydään läpi XNA:n ominaisuuksia luentomonisteen tapaisesti vaiheittaisen peliesimerkin nojalla.
English abstract: XNA was created by Microsoft and released in 2006 as an layer of abstraction to make programming games easier. It can be used to create games for Microsoft Windows, Xbox 360 and Zune media player. This thesis is built around an example game in which new features are added to and explained in each chapter.
Avainsanat: tietotekniikka, Microsoft, XNA, C#, peliohjelmointi, peli, ohjelmointi, content-järjestelmä
Keywords: information technology, Microsoft, C#, game, programming, content pipeline

Termiluettelo

Tälle sivulle on koottu yleisimpiä monisteessa käytettyjä teknisiä termejä.

Asset on yleisnimitys peliin kuuluvasta sisällöstä, joka käsittää tekstuurit, äänet, 3d-mallit ja muun pelin datan. XNA:ssa assettien paikka on projektin Content-kansio. Tässä tutkielmassa käytetään assetille suomennosta sisältökomponentti.

Efekti koostuu verteksi- ja pikselivarjostimista. 3d-mallien osat (*meshit*) piirretään aina efektejä käyttäen. Efektien ohjelmoimiseen XNA:ssa ja DirectX:ssä käytetään erillistä HLSL-ohjelmointikieltä (Reed 2009, s. 275–277).

Pikseli on piste näyttöruudulla.

Pikselivarjostin (engl. *pixel shader*) on näytönohjaimen ajama (ali)ohjelma, joka suoritetaan kerran jokaista näytöllä olevaa pikseliä kohden. Pikselivarjostin muuttaa pikselin väriä ja mahdollisesti muita ominaisuuksia sen paikan ja muiden tilatietojen perusteella. (Reed 2009, s. 275–277)

Sprite on kuva, joka piirretään enemmän tai vähemmän sellaisenaan ruudulle. Esimerkkejä spriteistä: pelaajahahmon kuva, vihollisen kuva, käyttöliittymäelementit, 3d-pelin kiikaritähtäin. (Grootjans 2009, s. 174)

Solution on kokoelma projekteja Visual Studiossa. Solution-tiedoston tarkenne on uudemmissa Visual Studioissa `.sln` ja vanhemmissa `.dsw`.

Tekstuuri on kuva, jota käytetään pintamateriaalina 3d-malleille (Grootjans 2009, s. 174). Arkikielessä tekstuurilla voidaan tarkoittaa myös spriteä, ja esimerkiksi luokkaa `Texture2D` käytetään XNA:ssa myös spritejen säilömiseen.

Verteksi on piste 3d-avaruudessa.

Verteksivarjostin (engl. *vertex shader*) on näytönohjaimen ajama (ali)ohjelma, joka suoritetaan kerran jokaista näkyvissä olevaa verteksiä kohden. Verteksivarjostin voi muuttaa olemassaolevien verteksien ominaisuuksia kuten paikkoja, värejä ja tekstuurikoordinaatteja. (Reed 2009, s. 275–277).

Sisältö

1 Lukijalle

Tämä moniste on tarkoitettu kandidaatintutkielmaksi Jyväskylän yliopistolle syyslukukaudeksi 2010-2011. Monistetta voidaan käyttää materiaalina (peli)ohjelmoinnin alkeiskursseilla.

Pohjatietona lukijalta edellytetään olio-ohjelmoinnin ja C#-ohjelmointikielen alkeet. Tietokonegrafiikan tai peliohjelmoinnin tuntemuksesta voi myös olla apua, mutta kumpikaan ei ole välttämätöntä asioiden ymmärtämiseksi.

Monisteen ensimmäinen osa (luvut 3-4) käsittelee itse XNA-kirjastoa, tarkentaen mikä XNA on, mitä sillä voi tehdä ja miltä tyypillinen XNA-peli näyttää.

Monisteen jälkimmäinen osa (luvut 5-10) käsittelee pelin tekemistä osa-alueittain esimerkkipelin nojalla. Luvussa 5 tutustutaan XNA:n 2d-grafiikkaominaisuuksiin ja piirretään tekstuureita ruudulle. Pelaaja otetaan mukaan toimintaan luvussa 6, jossa aiheena on ohjainten kuuntelu. Luvussa 7 herätetään viholliset ”henkiin” tekoälyn avulla ja laitetaan pelioliot ampumaan ja törmäilemään. Taustamusiikki ja äänet edellämainitulle. toiminnalle lisätään luvussa 8. Viimeisessä luvussa 9 tutustutaan lyhyesti 3d-grafiikkaan XNA:lla.

2 Visual Studiosta ja C#-ohjelmointikielestä

Tämä moniste on kirjoitettu aloittelevalle (peli)ohjelmoijalle, jolla on hallussa perusteet C#-kielestä ja Visual Studion käytöstä. Aiheen rajauksesta johtuen kielen syntaksiin tai rakenteisiin ei mennä tarkemmin, joten oheislukemistoksi suositellaan vapaavalintaista C#-opasta, jos kielen kanssa on epävarmuutta.

XNA-pelin tekeminen on mahdollista myös muilla .NET-kielillä, kuten Visual Basic ja Visual C++. Kaikki esimerkit ovat täysin toteutettavissa edellämainituilla kielillä, joten valinta perustuukin lähinnä ohjelmoijan omiin syntaktisiin mieltymyksiin.

3 XNA lyhyesti

Tässä luvussa määrittelemme, mikä XNA oikeastaan on ja millaisia pelejä sillä voidaan ohjelmoida menemättä vielä tarkempiin teknisiin yksityiskohtiin.

3.1 Mikä XNA on ja mitä sillä voidaan tehdä

XNA Game Studio on Microsoftin vuonna 2006 julkaisema kokoelma työkaluja peliohjelmointiin Windows-käyttöjärjestelmälle, Xbox 360 -pelikonsolille ja Zune-mediasoittimelle. Sen merkittävin osa on XNA Framework (tästä eteenpäin XNA), joka yhdessä .NET Frameworkin kanssa toimii välikerroksena ohjelmoijan ja DirectX-peliohjelmointirajapinnan välillä. Näin ollen ohjelmoijan tarvitsee kiinnittää vähemmän huomiota laitteistoteknisiin yksityiskohtiin peliä tehdessään. (MSDN 2009).

3.2 Mitä XNA-pelien ohjelmointiin vaaditaan

Vaikka XNA Game Studio -paketti onkin täysin ilmainen, ainoa virallisesti tuettu kehitysympäristö XNA-peliprojekteille on Microsoftin oma Visual Studio. Maksullinen Professional-versio ei ole kuitenkaan välttämätön, vaan myös ilmaisella Express-versiolla pelin tekeminen onnistuu. Express-versio on ladattavissa Microsoftin sivustolta maksutonta rekisteröitymistä vastaan. Visual Studio sisältää myös pelien ajamiseen tarvittavan .NET Frameworkin, XNA Game Studio on asennettava erikseen. (Reed 2009, s. 1)

3.3 Mitä XNA-pelien pelaamiseen vaaditaan

XNA-pelin käynnistämiseen Windows-ympäristössä tarvitaan vähintään sama .NET Frameworkin ja XNA Game Studion versio, kuin millä peli on tehty, sekä DirectX 9 tai uudempi. Visual Studiota ei enää tarvita, kun peli on käännetty. Pelin kansiosta löytyy itse exe-tiedoston ja mahdollisten dll-tiedostojen lisäksi alikansio "Content", johon on tallennettu pelissä tarvittava sisältö XNA:n omassa binäärimuodossa. Laitteistovaatimuksena on näytönohjain vähintään Shader Model 1.1 -tuella, tosin laajempia 3d-ominaisuuksia käyttävät pelit voivat vaatia uudempaakin versiota. (Carter 2009, s. 9–10)

XNA-pelin pelaaminen Xbox 360:lla vaatii vuosimaksullisen XNA Creators Club Premium -jäsenyyden. Jäsenyyteen sisältyy mahdollisuus pelata ja arvostella toisten ohjelmoijien tekemiä pelejä ja laittaa tekemänsä pelit myyntiin Xbox LIVE -kauppapaikkaan (Carter 2009, s. 10). Pelin siirtäminen laitteeseen onnistuu verkkokaapelilla XNA Game Studio Device Centerin kautta (käynnistysvalikosta). (Grootjans 2009, s. 5–6)

Pelin siirtäminen Zuneen onnistuu myös Device Centerin kautta USB-kaapelia pitkin. Siirron jälkeen pelin pitäisi ilmestyä laitteen Games-valikkoon, olettaen että virheitä ei tapahdu ja laitteen firmware on ajan tasalla. (Grootjans 2009, s. 8–9)

3.4 Mitä XNA-pelien myymiseen vaaditaan

XNA:n käyttöoikeussopimus (EULA) ei erikseen kiellä omatekoisten PC-pelien myymistä, mutta asettaa sille tiettyjä rajoituksia. Kaupallisissa peleissä ei saa käyttää Windows Live-palvelua (XNA EULA 3.1, 3e,iv), ja pelaajien väliseen ääni- ja tekstikommunikaatioon on käytettävä XNA:n omia rutiineja ennalta määrätyillä parametreilla (XNA EULA 3.1, 2e). Zune-pelien myyminen on kokonaan kielletty (XNA EULA 3.1, 2c,i).

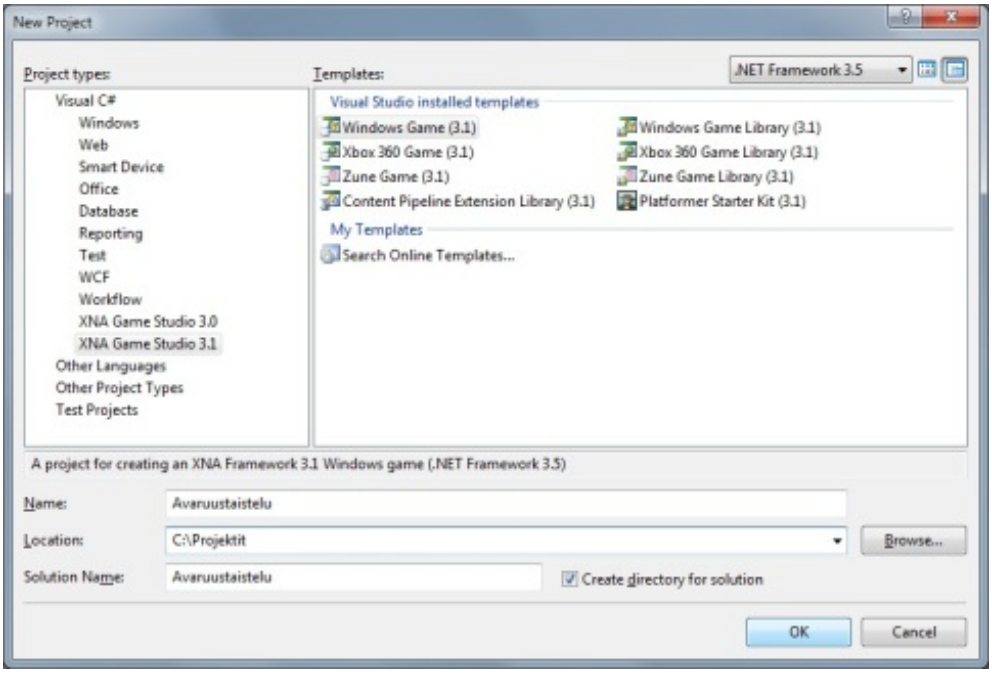
Xbox 360 -pelien myymisen itse EULA kieltää kokonaan, mutta se on mahdollista Microsoftin Xbox Live -kauppapaikan kautta (XNA EULA 3.1, 2b,i). Pelin julkaisemiseen Xbox Live Marketplacessa vaaditaan Xbox LIVE Silver tai Gold -jäsenyyys (XNA EULA 3.1, 2b,ii) sekä vuosimaksullinen XNA Creators Club Premium -jäsenyys, ja itse pelin on läpäistävä vertaisarviointi. Kun peli on hyväksytty vertaisarvioinnissa ja pelin tekijä on antanut Microsoftille henkilö- ja verotietonsa, peli ilmestyy kauppapaikkaan käyttäjien saataville. Microsoftin osuus tuotoista on vähintään 30%, tai enemmän jos peli osoittautuu erityisen suosituksi. Pelin tekijälle tai tekijöille maksetaan verojen ja Microsoftin osuuden jälkeen jäävä summa tilille neljännesvuosittain, jos se on vähintään 150 dollaria. (Grootjans 2009, s. 727–737)

4 XNA-pelin rakenne

Tässä luvussa tutustumme Visual Studio -ympäristöön ja aloitamme esimerkkipelin tekemisen luomalla uuden projektin ja tarkastemalla sen sisältöä.

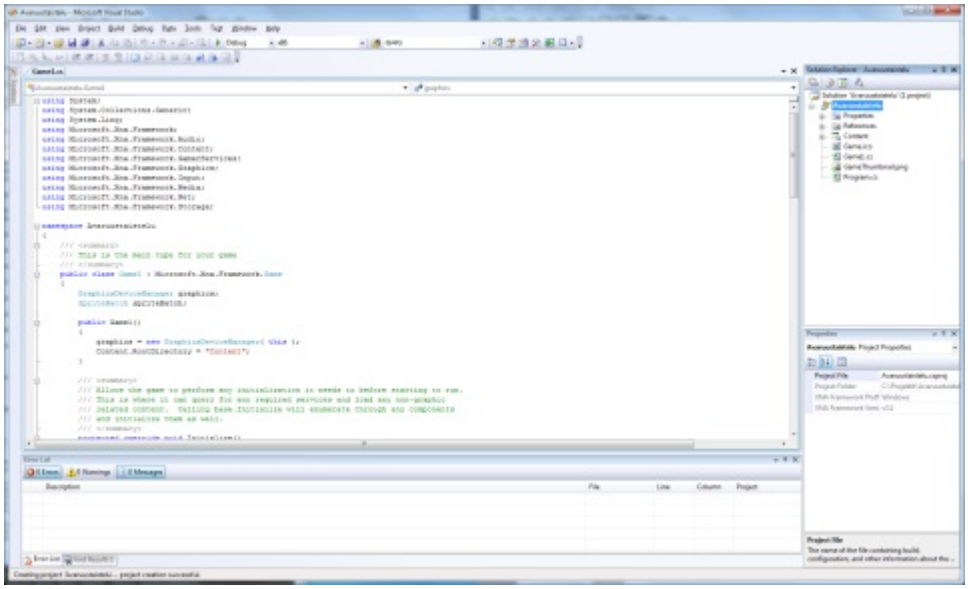
4.1 Projektin luominen

Luodaan uusi peliprojekti nimellä Avaruustaistelu valitsemalla File->New->Project, jolloin saadaan seuraavanlainen valintaikkuna:



Kuva 1: Uuden projektin luominen Visual Studio 2008 Professional -ympäristössä.

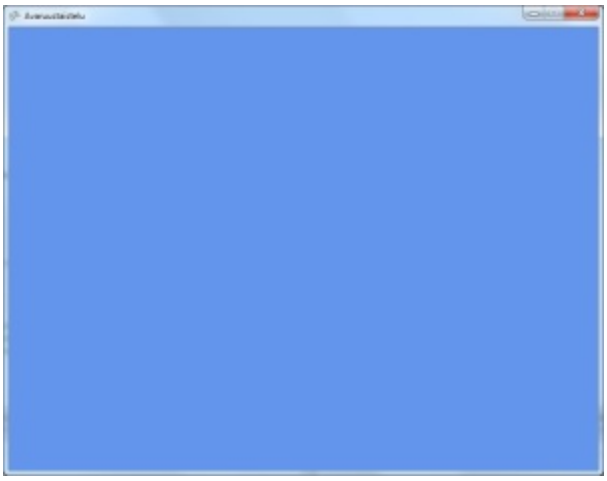
Kun XNA-peliprojekti on luotu, aukeaa eteen uusi tiedosto Game1.cs ja joukko muita tiedostoja ja kansioita oikealla näkyvässä Solution Explorer -näkymässä:



Kuva 2: Visual Studio 2008 -ohjelmointiympäristö.

Projektin mukana luotu tiedosto Game1.cs sisältää luokan Game1, johon pelilogiikka olisi tarkoitus ohjelmoida. Ennen kuin perehdymme tarkemmin luokan

metodeihin, kokeillaanpa ajaa peli ja katsotaan, mitä tapahtuu. Pikanäppäin ohjelman käynnistykseen Visual Studiossa on F5.



Kuva 3: Projektimallista luotu XNA-peli. Ohjelmakoodiin ei ole koskettu vielä tässä vaiheessa.

4.2 Peliluokka ja sen metodit

Toisin kuin esimerkiksi konsoli- tai OpenGL-ohjelmoinnissa, tyypillisessä XNA-pelissä main-metodiin ei tarvitse koskea. Metodi löytyy kyllä `Program.cs` -tiedostosta, mutta sen ainoa tarkoitus on luoda peliluokka ja ohjata suoritus sen Run-metodiin:

```
/// <summary>
/// The main entry point for the application.
/// </summary>
static void Main( string[] args )
{
    using ( Game1 game = new Game1() )
    {
        game.Run();
    }
}
```

Tulemme myös huomaamaan eron tapahtumapainotteiseen komponenttiohjelmointiin (Windows Forms ym.). Oletuksena XNA:ssa ei ole valmiita tapahtumia, mutta sellaisten tekeminen itse on mahdollista (Hejlsberg & Wiltamuth ja Golde 2006, s. 327). Myös eräät laajennuskirjastot, kuten Jypeli (Jyväskylän yliopisto 2010), tarjoavat valmiita tapahtumia esimerkiksi näppäimistön ja peliohjainten kuunte-

luun. Omien tapahtumien tekeminen kuitenkin hankaloittaa aihetta vielä tässä vaiheessa, ja on siksi jätetty tämän monisteen ulkopuolelle.

XNA:ssa tärkein luokka on peliluokka, joka luodaan projektin yhteydessä nimellä `Game1`. Luokka peritään XNA:n valmiista yliluokasta `Game`, joka sisältää esimerkiksi aiemmin mainitun `Run`-metodin ja monia muita. Jokaista peliä varten on täsmälleen yksi peliluokka.

`Initialize`-metodissa nimensä mukaisesti alustetaan peli. Alustuksiin kuuluu yleensä pelikentän ja -hahmojen lataaminen, alkuarvojen asetus ja kaikki muu toiminta, mikä on tarkoitus suorittaa (kerran) ennen pelin alkua. Myös peliluokan rakentajassa voidaan tehdä joitakin alustuksia, mutta on huomattava, etteivät kaikki peliluokan ominaisuudet ole vielä käytettävissä. Oikein laadittuna `Initialize`a voidaan myös kutsua myöhemmin muualta pelistä, jos halutaan aloittaa peli alusta.

`LoadContent` on tarkoitettu tekstuurien, äänien ja muun sisällön lataamiseen. `LoadContent` suoritetaan vain kerran pelin aikana, eikä sitä kuulu kutsua itse missään vaiheessa. Metodin käyttö muuhun kuin sisällön lataamiseen on ”sallittua” lähinnä alustuskoodille, joka käyttää ladattua sisältöä tai muuten ei sovi rakentajaan tai `Initialize`-metodiin.

`UnloadContent`-metodia kutsutaan automaattisesti pelin loputtua. XNA osaa vapauttaa itse kaiken `Content.Load`-metodilla (ks. seuraava luku) ladatun sisällön, joten tätä metodia tarvitaan vain niissä harvoissa tapauksissa, joissa sisältöä on ladattu jollain muulla tavalla tai käytössä on olioita, joiden käyttämä muisti on erikseen vapautettava kutsumalla niiden `Dispose`-metodia.

`Update` on metodi pelilogiikan päivittämiseen, eli käytännössä sanoen kaikkeen toimintaan joka ei liity ruudulle piirtämiseen. XNA pyrkii kutsumaan metodia 60 kertaa sekunnissa. Päivitystaajuutta voi myös säätää itse jos halutaan parantaa vasteaikoja tai vähentää kuormitusta.

`Draw`-metodi on tarkoitettu 2d- ja 3d-grafiikan piirtämiseen ruudulle. XNA pyrkii kutsumaan metodia näytön (pystysuuntaisen) virkistystaajuuden kanssa samaan tahtiin. Raskaampi piirtämiseen liittyvä laskenta kuten 3d-maailman kameran matriisit suositellaan kuitenkin tekemään `Update`-metodissa.

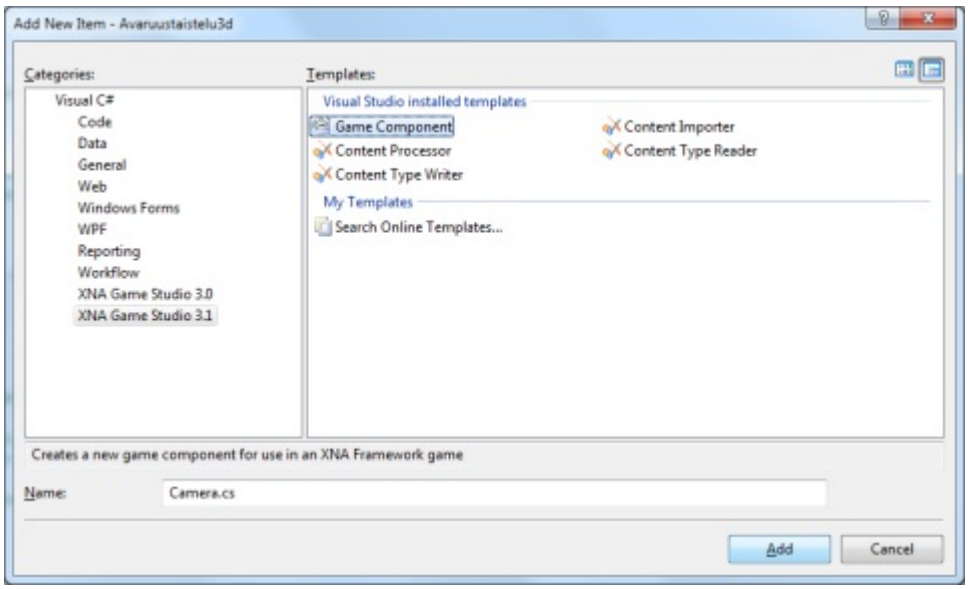
(Grootjans 2009, s. 4)

4.3 Komponentit

Kaikkea pelin päivitykseen ja piirtämiseen liittyvää logiikkaa ei kuitenkaan kannata kirjoittaa suoraan pääluokan metodeihin, sillä ennen pitkää koodimäärä kasvaa ja muuttuu mahdottomaksi hallita. Kokonaisuudet, kuten peliolioiden liikuttelu ja

törmäyskäsittely onkin tapana eristää omaan luokkaansa ja kutsua niiden vastaavia metodeja. XNA:ssa tätä helpottamaan on tehty erityinen luokka `GameComponent`, joista perittyjä omia luokkia sanotaan komponenteiksi. (Reed 2009, s. 67)

`GameComponent`-luokassa on rakentajan lisäksi valmiina metodit `Initialize` ja `Update`, joita kutsutaan automaattisesti, kun komponentti on lisätty peliin. Piirrettäville komponenteille voidaan käyttää kantaluokkana `DrawableGameComponent`-luokkaa, jossa on valmiina myös metodi `Draw`. Peliin voidaan lisätä uusi komponentti projektin kontekstivalikosta (Project Explorer -näkymä) valitsemalla *Add->New Item...* ja valitsemalla avautuvasta ikkunasta *XNA Game Studio 3.1*-kohdan alta *Game Component*. (Grootjans 2009, s. 15)



Kuva 4: Lisätään projektiin uusi komponenentti Camera.

Piirrettävää komponenttia ei version 3.1 listasta valmiina löydy, mutta sellaisen voi tehdä tavallisesta komponentista muuttamalla sen periytymään luokasta `DrawableGameComponent` luokan `GameComponent` sijaan ja lisäämällä metodin `public override void Draw`. (Grootjans 2009, s. 15)

Kun komponentti on valmis, voidaan se lisätä peliin. Tämän jälkeen komponentti päivitetään ja piirretään automaattisesti ilman erillisiä kutsuja. (Reed 2009, s. 67)

```
Camera camera;

protected override void Initialize()
{
    camera = new Camera( this );
```

```
        Components.Add( camera );

        // ...
}
```

4.4 Sisältöä peliin — johdatus content-järjestelmään

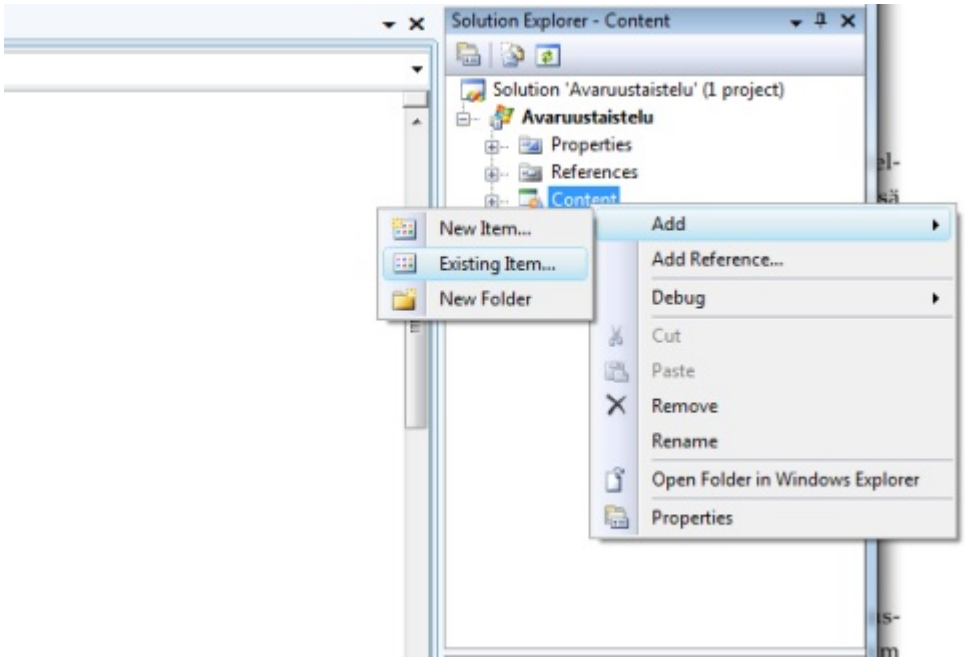
Yksi suurimpia ongelmia peliohjelmoinnissa on tiedostoformaattien valinta. Käyttääkö JPEG- vai PNG-kuvia? Pitääkö ääniefektit WAV-muodossa vai pakkaisiko vaikkapa Ogg Vorbisilla? Valinnoista riippumatta jokainen formaatti vaatii oman kirjastonsa, joilla puolestaan kaikilla on omat lisenssiehtonsa ja riippuvuutensa joista tulee ottaa selvää ennen käyttöä.

Content-järjestelmä on XNA:n vastaus ongelmaan. Yksinkertaistettuna järjestelmä ottaa vastaan tunnetuissa muodoissa olevia tiedostoja ja tekee niistä pelissä käytettäviä sisältökomponentteja (asset). Komponentit käännetään automaattisesti XNA:n käyttämään binäärimuotoon ja kopioidaan oikeaan hakemistoon. Valmiiksi tuettuja tiedostomuotoja ovat: (Carter (2009, s. 114), Miles (2009, s. 125) ja MSDN (2009))

- 3d-mallit: .x ja .fbx
- Fontit: .spritefont (XML-kääre TrueType- ja OpenType-fonteille)
- Musiikki: .mp3, .wma
- Tekstuurit: .bmp, .dds, .dib, .hdr, .jpg, .pfm, .png, .ppm ja .tga
- Äänet: .wav, .xap (XACT-projekti)

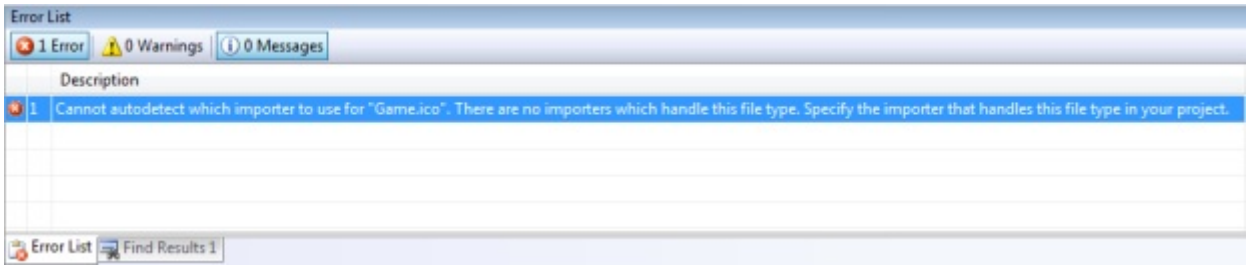
Content-järjestelmää on myös mahdollista laajentaa tukemaan muita tiedostomuotoja (custom content importer), sekä käyttämään erilaisia muunnoksia ja suotimia (custom content processor). Lähteet Carter (2009) ja Grootjans (2009) tarjoavat erinomaisia esimerkkejä kiinnostuneille.

Sisältökomponentin eli assetin lisääminen projektiin onnistuu valitsemalla Solution Explorerista kohta *Content* ja sen kontekstivalikosta (oikean hiirenpainikkeen takana oleva valikko) *Add -> Existing Item*:



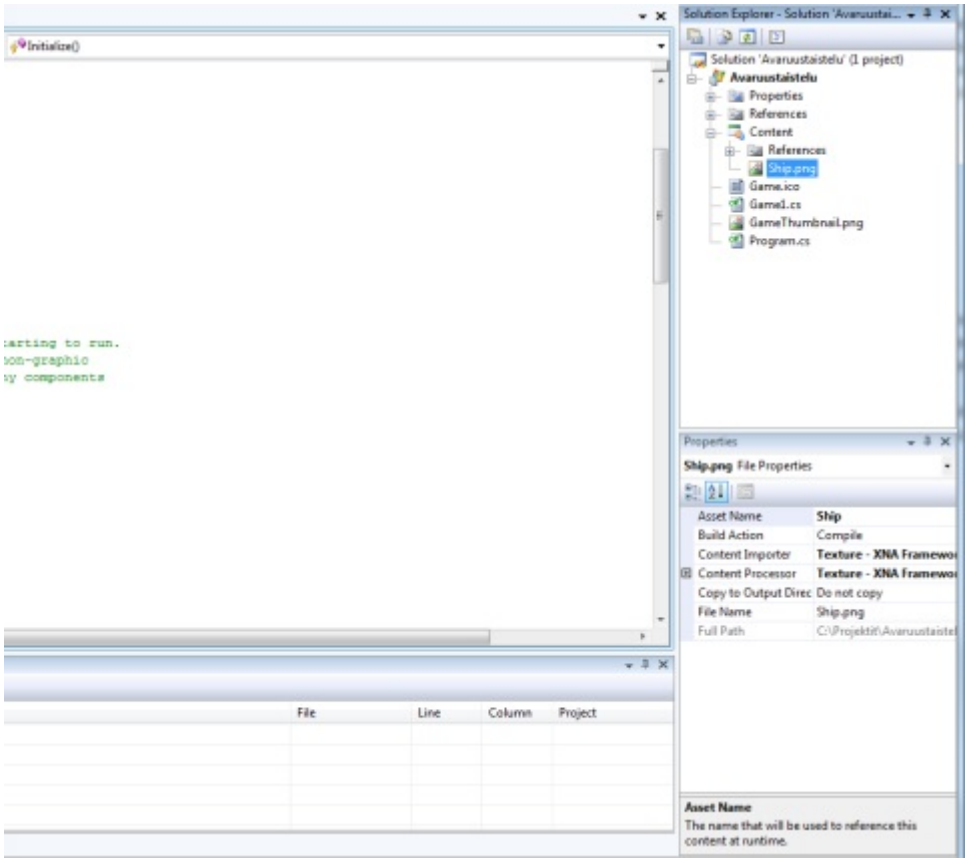
Kuva 5: Uuden tiedoston lisääminen content-puuhun.

Ilmestyvästä tiedostonvalintaikkunasta voi valita vapaasti minkä tahansa tiedoston, ja XNA ottaa sen automaattisesti käyttöön. Jos tiedostotyyppi ei ole tuettu, seuraa virheilmoitus käännettäessä projektia:



Kuva 6: Virheilmoitus: tiedostomuoto ei ole tuettu.

Lisätään kuvatiedosto "ship.png". Nyt kuvaa voidaan käyttää esimerkiksi tekstuurina.



Kuva 7: Lisätty kuva ship.png. Näkyvissä myös sisältökomponentin nimi ja ominaisuudet.

5 2d-grafiikka

Ensimmäinen askel (ainakin ensimmäisen) pelin tekemisessä on yleensä saada jotain näkymään ruudulla. Tämän luvun ensimmäisessä alaluvussa lataamme lisäämämme kuvatiedoston tekstuuriksi ja toisessa piirrämme sen. Kolmannessa alaluvussa luomme oman peliolion tekstuurin ja muiden ominaisuuksien yhteen kokoamiseksi.

5.1 Tekstuurit ja niiden lataaminen

2d-grafiikassa peruselementti on tekstuuri, jota vastaava luokka XNA:ssa on `Texture2D`. Viime luvussa lisäsimme projektiin tiedoston *Ship.png*, jonka XNA tunnisti tyypin mukaan automaattisesti tekstuuriksi. Tehdään nyt luokalle attribuutti `shipTex`, johon tekstuuri ladataan:

```
public class Game1 : Microsoft.Xna.Framework.Game
{
  GraphicsDeviceManager graphics;
  SpriteBatch spriteBatch;
  Texture2D shipTex;

  public Game1()
  {
```

Tekstuurin lataaminen muuttujaan tapahtuu XNA:n `Game`-luokasta perityn `Content`-olion `Load`-metodia kutsumalla. Metodi ottaa tyyppiparametrin kulmasuluissa ja asset-nimen normaalina parametrina. Ensimmäinen on tässä tapauksessa `Texture2D` ja jälkimmäinen `Ship`. Metodi palauttaa `Texture2D`-tyyppisen olion, joka voidaan sijoittaa luomaamme attribuuttiin myöhempää käyttöä varten. Tämä kaikki supistuu yhteen riviin `LoadContent`-metodissa:

```
shipTex = Content.Load<Texture2D>( "Ship" );
```

Lisättyämme latausrivin ja poistettuamme valmiit kommentit `LoadContent` näyttää nyt tältä:

```
protected override void LoadContent()
{
  spriteBatch = new SpriteBatch( GraphicsDevice );
  shipTex = Content.Load<Texture2D>( "Ship" );
}
```

5.2 Tekstuurin piirtäminen näyttöruudulle

Jos kokeilemme nyt ajaa ohjelman, mikään ei näytä muuttuneen. Tähän mennessä olemme vasta ladanneet tekstuurin muuttujaan, se täytyy myös piirtää ruudulle jos sen haluaa näkyviin. Piirtäminen tapahtuu `Draw`-metodissa, ja siihen voidaan käyttää `LoadContent`issa alustettua oliota `spriteBatch`.

```
protected override void Draw( GameTime gameTime )
{
  GraphicsDevice.Clear( Color.CornflowerBlue );
  spriteBatch.Begin();
  spriteBatch.Draw( shipTex, Vector2.Zero, Color.White );
  spriteBatch.End();
```

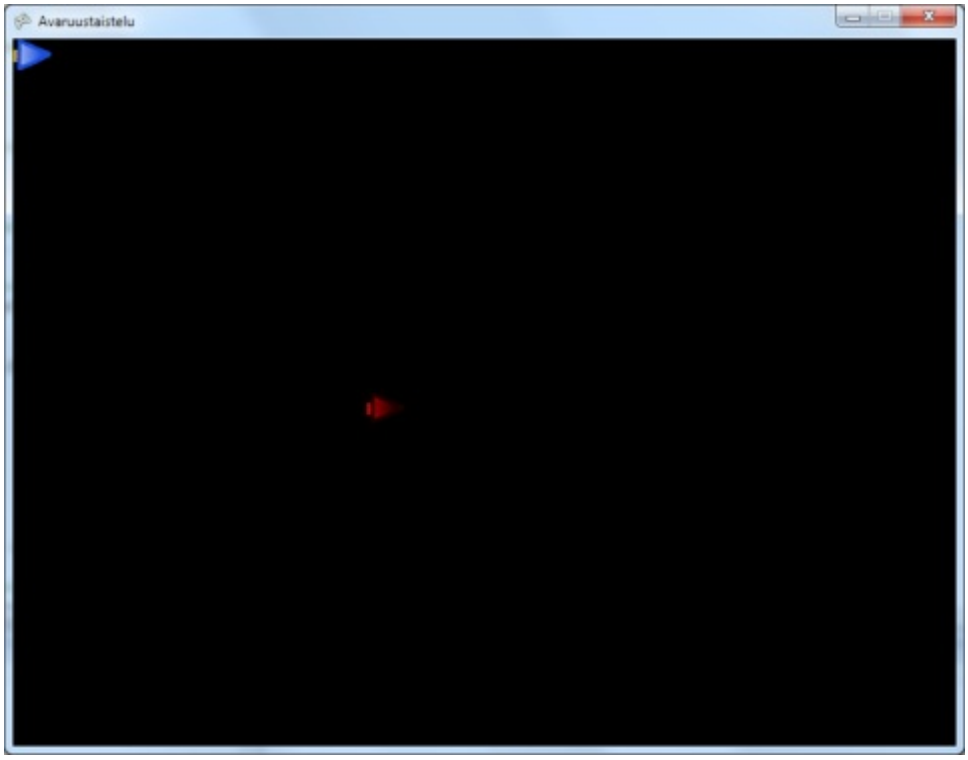
```
    base.Draw( gameTime );
}
```

Ylläoleva ohjelmakoodi piirtää tekstuurin shipTex ruudun vasempaan yläkulmaan, jota vastaa nollavektori `Vector.Zero`. Koordinaatiston X-koordinaatti kasvaa oikealle ja Y-koordinaatti alaspäin, kuten ruutukoordinaateille on tyypillistä. Valitusta pisteestä tulee oletuksena tekstuurin vasen yläreuna, eli toisin sanoen kuvan origo on sen vasemmassa yläreunassa (Carter 2009, s. 172).

`Draw`-metodin kolmas parametri on väri, jolla tekstuuri "sävytetään"eli käytännössä kerrotaan tekstuurin jokaisen pikselin RGB-arvot komponenteittain. Koska `Color.White` vastaa RGB-arvoa (1,1,1), pysyvät tekstuurin värit ennallaan. `Draw`-metodista on lukuisia ylikuormitettuja versioita, joilla tekstuureita voidaan piirtää mm. kulmassa tai osittain. (Grootjans 2009, s. 177)

Ei ole mitään syytä, miksi samaa tekstuuria ei voisi käyttää myös useampaan kertaan. Lisätään vielä havainnollisuuden vuoksi toinen `Draw`-kutsu ensimmäisen jälkeen ja katsotaan miltä peli näyttää sen jälkeen. Muutetaan samalla taustaväri paremmin avaruutta vastaavaksi mustaksi:

```
protected override void Draw( GameTime gameTime )
{
    GraphicsDevice.Clear( Color.Black );
    spriteBatch.Begin();
    spriteBatch.Draw( shipTex, Vector2.Zero, Color.White );
    spriteBatch.Draw( shipTex, new Vector2( 300, 300 ),
                      Color.Red );
    spriteBatch.End();
    base.Draw( gameTime );
}
```



Kuva 8: Kaksi alusta avaruudessa.

On tärkeää, että kaikki `Draw`-kutsut sijoitetaan `Begin`- ja `End`-kutsujen väliin, sillä se valmistaa näytönohjaimen ottamaan vastaan juuri tekstuureja, eikä esimerkiksi verteksejä. (Reed 2009, s. 19)

5.3 Pelioliot

Yleensä tekstuuri itsessään on vain jonkin peliolion, kuten tässä tapauksessa avaruusaluksen, näkyvä ominaisuus. Pelioliolla voi olla myös muita ominaisuuksia: paikka, nopeus, osumapisteet, ammuksien määrä jne. Jos pelissä tulee olemaan useita erityyppisiä peliolioita, on järkevää tehdä niille yhteinen kantaluokka.

Luokka voidaan tehdä Visual Studiossa valitsemalla projektin kontekstivalikosta *Add -> Class* ja antamalla luokalle nimi.

```
public class GameObject
{
  public Vector2 Position { get; set; }
  public float Speed { get; set; }
  public float Angle { get; set; }
  public Texture2D Texture { get; set; }
```

```
  public virtual void Update( GameTime gameTime )
  {
      double dt = gameTime.ElapsedGameTime.TotalSeconds;

      if ( dt <= 0 )
        return;

      double dx = Speed * Math.Cos( Angle ) / dt;
      double dy = Speed * Math.Sin( Angle ) / dt;

      Position += new Vector2( (float)dx, (float)dy );
  }

  public virtual void Draw( SpriteBatch spriteBatch )
  {
      Rectangle srcRect =
        new Rectangle( 0, 0, Texture.Width, Texture.Height );
      Vector2 origin =
        new Vector2( Texture.Width / 2, Texture.Height / 2 );

      spriteBatch.Draw(
        Texture, Position, srcRect,
        Color.White, Angle, origin, 1,
        SpriteEffects.None, 0 );
  }
}
```

Metodissa `Update` liikutetaan pelioliota sen mukaan, mikä sen nopeus ja rintamasuunta on. `gameTime.ElapsedGameTime.Seconds` kertoo, kuinka monta sekuntia on kulunut aikaisemmasta `Update`-metodin kutsusta (Reed 2009, s. 37), ja tällä ajalla jakamalla saadaan matka kummankin komponentin suuntaan niin, että tekstuuri näyttää liikkuvan tasaisesti. `Draw`-metodiin on siirretty piirtokoodi pelin vastaavasta metodista sillä muutoksella, että piirtämiseen käytetään myös kulmaa. Kummatkin metodit ovat virtuaalisia, mikä tarkoittaa että perityt luokat voivat halutessaan ylikirjoittaa (engl. *override*) metodin, eli käyttää sen tilalla omaa samannimistä metodia ja kutsua tai olla kutsumatta alkuperäistä (Hejlsberg & Wiltamuth

ja Golde 2006, s. 311). Tätä tarvitaan myöhemmin esimerkissämme, kun vihollisille lisätään tekoäly.

Nyt voimme lisätä peliluokkaan attribuutit aluksille ja alustaa ne `LoadContent`-metodissa.

```
GameObject playerShip;
GameObject enemyShip;
```

```
protected override void LoadContent()
{
  spriteBatch = new SpriteBatch( GraphicsDevice );
  shipTex = Content.Load<Texture2D>( "Ship" );

  playerShip = new GameObject();
  playerShip.Texture = shipTex;
  playerShip.Position = new Vector2( 100, 300 );

  enemyShip = new GameObject();
  enemyShip.Texture = shipTex;
  enemyShip.Position = new Vector2( 600, 300 );
  enemyShip.Angle = MathHelper.Pi;
}
```

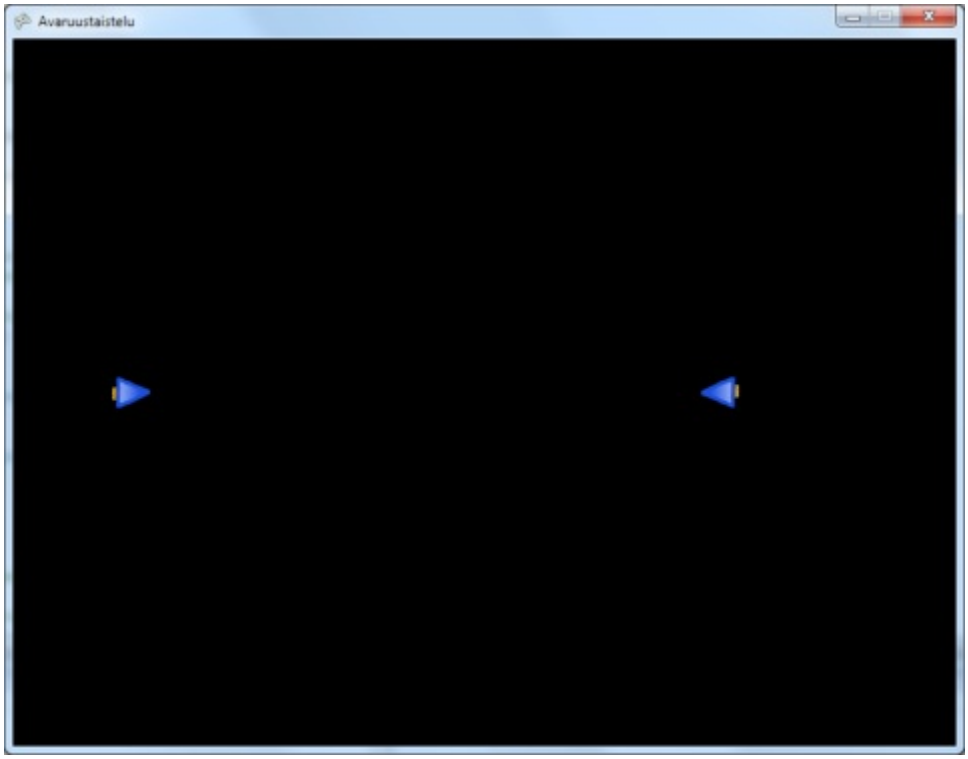
Koska pelioliomme sisältää oman piirtometodinsa, sitä tarvitsee vain kutsua peliluokan `Draw`-metodissa.

```
protected override void Draw( GameTime gameTime )
{
    GraphicsDevice.Clear( Color.Black );
    spriteBatch.Begin();
    playerShip.Draw( spriteBatch );
    enemyShip.Draw( spriteBatch );
    spriteBatch.End();
    base.Draw( gameTime );
}
```

Huomaa, kuinka peliolioiden `Draw`-metodi saa `spriteBatch`in parametrina sen sijaan, että jokainen käyttäisi omaa oliotaan piirtämiseen. Näin kaikki piirtokutsut jäävät yksien `Begin`- ja `End`-kutsujen väliin, mikä tietää vähemmän työtä näy-

tönohjaimelle (Carter 2009, s. 173). Suuremmilla määrillä tekstuureja nopeusero on merkittävä.

Samaan tapaan lisätään myös `Update`-metodien kutsu pelin `Update`-metodiin, jossa parametriksi tulee `gameTime`. Peliluokka on nyt huomattavasti siistimpi, ja myös alusten ominaisuudet ovat paremmin muutettavissa. Yleiskäyttöisyytensä ansiosta pelioliota voidaan käyttää myös muissa projekteissa.



Kuva 9: Alukset ottamassa mittaa toisistaan.

6 Peliohjaimet ja interaktio

Nyt kun olemme saaneet ruudulle aluksia, on aika panna ne liikkeelle. Tässä luvussa päästämme pelaajan ohjaamaan toista aluksista ja samalla tutustumme XNA:n tapaan kuunnella ohjainlaitteita.

6.1 Ohjainten kuuntelusta yleisesti

XNA:ssa kaikkien ohjainten kuuntelu tehdään pollaamalla, eli tarkistamalla säännöllisesti, onko ohjain tietyssä tilassa. Nämä tarkistukset on luontevaa tehdä pelin (tai jonkun sen komponentin) `Update`-metodissa. Jokaisessa ohjainluokassa on meto-

di `GetState`, jota kutsumalla saadaan tilaolio, joka kertoo ohjaimen senhetkisen tilan. Tila voidaan myös tallettaa muuttujaan, jolloin näppäimien pohjassa pitämistä voidaan tarkastella vertailemalla tiloja keskenään. (Reed 2009, s. 40–50)

Peliohjainten osalta XNA tukee yhtäaikaisesti 1-4 langallista tai langatonta Xbox 360 -ohjainta (Carter 2009, s. 93). Jälkimmäiset toimivat sekä PC:llä että Xbox 360:lla. Zunen ohjaimet tunnistuvat Xbox 360 -ohjaimena, ja niiden tiloja voi lukea samaan tapaan (Reed 2009, s. 165–166). Natiivitukea yleisille peliohjaimille (joystickit, ei-Xbox-padit, ratit jne.) XNA:ssa ei ole.

6.2 Näppäimistö

Näppäimistön tila luetaan metodilla `Keyboard.GetState()`. Paluuarvona on `KeyboardState`-tyyppinen olio, jonka metodeita `IsKeyUp`, `IsKeyDown` ja `GetPressedKeys` kutsumalla voidaan saada tietoa näppäinten tilasta. (Carter 2009, s. 89)

Seuraava ohjelmakoodi kääntää pelaajan alusta, kun vasenta tai oikeaa nuolinäppäintä painetaan.

```
KeyboardState kbstate = Keyboard.GetState();

if ( kbstate.IsKeyDown( Keys.Left ) )
  playerShip.Angle -= MathHelper.Pi / 60;
if ( kbstate.IsKeyDown( Keys.Right ) )
  playerShip.Angle += MathHelper.Pi / 60;
```

Olisi myös täysin syntaktisesti oikein olla tallentamatta näppäimistön tilaa muuttujaan ja käyttää ehtoja `Keyboard.GetState().IsKeyDown( Keys.Left )` ja `Keyboard.GetState().IsKeyDown( Keys.Right )`, mutta silloin näppäimistön tila tarkistettaisiin kahteen kertaan. Varsinkin useampien ehtojen tapauksessa tämä hidastaa peliä ja saattaa aiheuttaa ongelmia, ja siten luetaan huonoksi ohjelmointityyliksi. (Reed 2009, s. 45)

Nyt alus pyörii paikallaan. Seuraava tehtävä olisi saada se liikkumaan eteenpäin. Jotta pelaaja ei tuhoaisi alustaan samointein pitämällä kaasua pohjassa liian pitkään, tehdään nopeuden säädöstä portaittainen. Tähän tarvitsemme tiedon näppäimistön aikaisemmasta tilasta, joka voidaan tallentaa peliluokan (yksityiseen) attribuuttiin.

```
KeyboardState oldKbState = Keyboard.GetState();
```

Tila alustetaan, jottei sille tarvitse tehdä erillistä null-tarkistusta. Lisätään aikaisempien kuunteluehtojen perään seuraavat rivit:

```
if ( oldKbState.IsKeyUp( Keys.Up ) &&
     kbstate.IsKeyDown( Keys.Up ) )
       playerShip.Speed += 0.02f;

if ( oldKbState.IsKeyUp( Keys.Down ) &&
    kbstate.IsKeyDown( Keys.Down ) )
      playerShip.Speed -= 0.02f;

oldKbState = kbstate;
```

Nyt aluksen nopeuteen lisätään 0.02 yksikköä sekunnissa joka kerta kun ylöspäin-nuolinäppäintä painetaan, ja vastaavasti vähennetään saman verran alaspäin-nuolinäppäimestä. Kirjain *f* vakion lopussa tarkoittaa, että luku on tyyppiä *float* oleva liukuluku.

6.3 Xbox 360 -peliohjaimen digitaaliset kontrollit

Xbox 360 -peliohjaimessa on kaksi analogista tattiohjainta, kaksi analogista liipaisinta sekä useita digitaalisia näppäimiä.

Kuva 10: Xbox 360 -ohjain. Punaisella merkittynä ovat `GamePadState`-luokan jäsenmuuttujat.

Tässä alaluvussa lisäämme peliimme mahdollisuuden liikuttaa alusta digitaalisen D-padin avulla. Luetaan ensin ohjaimen tila muuttujaan `Update`-metodissa.

```
GamePadState gpstate = GamePad.GetState( PlayerIndex.One );
```

`PlayerIndex.One` kertoo, että haluamme nimenomaan ensimmäisen pelaajan peliohjaimen tilan. Ohjaimia voi olla maksimissaan neljä yhtäaikaisesti, ja niiden järjestys määräytyy kytkemisjärjestyksen mukaan.

Digitaalisten näppäinten tilan tarkistus `GamePadState`-olion avulla tapahtuu hyvin pitkälle samaan tapaan kuin näppäimistön näppäinten tilan tarkistus `KeyboardState`a käyttäen. Ainoana merkittävänä erona peliohjaimen näppäin on `Button` siinä missä näppäimistön näppäin on `Key`. Seuraava ohjelmakoodi `Update`-metodissa mahdollistaa aluksen kääntämisen sekä näppäimistöllä että peliohjaimella:

```
if ( kbstate.IsKeyDown( Keys.Left ) ||
     gpstate.IsButtonDown( Buttons.DPadLeft ) )
       playerShip.Angle -= MathHelper.Pi / 60;

if ( kbstate.IsKeyDown( Keys.Right ) ||
     gpstate.IsButtonDown( Buttons.DPadRight ) )
       playerShip.Angle += MathHelper.Pi / 60;
```

Myös nopeuden säätäminen voidaan tehdä samaan tapaan kuin näppäimistöllä.

```
GamePadState oldGpState = GamePad.GetState(PlayerIndex.One);
```

```
if ( oldGpState.IsButtonUp( Buttons.DPadUp ) &&
     gpstate.IsButtonDown( Buttons.DPadUp ) )
       playerShip.Speed += 0.02f;

if ( oldGpState.IsButtonUp( Buttons.DPadDown ) &&
     gpstate.IsButtonDown( Buttons.DPadDown ) )
       playerShip.Speed -= 0.02f;

oldGpState = gpstate;
```

6.4 Xbox 360 -peliohjaimen analogiset kontrollit

Analogisten kontrollien tärkein etu pelaajalle digitaalisiin verrattuna on se, että peli reagoi sitä voimakkaammin, mitä enemmän kontrollia painaa. Ohjelmoija hyötyy analogisuudesta ehtolauseiden vähentymisenä. Esimerkkinä aluksen kääntämiseen vasemmalla tatilla riittää yksi lause `Update`-metodissa:

```
playerShip.Angle +=
  MathHelper.Pi / 60 * gpstate.ThumbSticks.Left.X;
```

Tattien X- ja Y-koordinaatit on määritelty liukuluvuiksi välille -1.0 – 1.0, missä negatiivinen 1 vastaa vasenta tai alareunaa ja positiivinen oikeaa tai yläreunaa. Liipaisimien vaihteluväli on 0.0 (irti) – 1.0 (täysin painettuna). (Reed 2009, s. 48)

6.5 Hiiren käyttäminen kursorin kanssa

Oletuksena hiiren kursoria ei näytetä pelissä, oletettavasti siksi että ohjelmoija voi halutessaan tehdä oman sprite-kursorin, joka seuraa hiiren koordinaatteja. Oletuskursori voidaan kuitenkin näyttää asettamalla `Initialize`-metodissa peliluokan attribuutin `IsMouseVisible` arvoksi `true`.

Myös hiirellä (`Mouse`) on metodi `GetState`, joka puolestaan palauttaa `MouseState`-tyyppisen olion. `MouseState` sisältää ominaisuudet `X` ja `Y` hiiren koordinaateille, `ScrollWheelValue` rullan asennolle sekä tilan jokaiselle näppäimelle. Näppäimien tilan voi tarkistaa `ButtonState`-enumeraation arvoja `Pressed` ja `Released` vastaan. Asetetaan aluksen kulma käyttäen näitä ominaisuuksia.

```
MouseState mstate = Mouse.GetState();

if ( mstate.LeftButton == ButtonState.Pressed )
{
  double dx = mstate.X - playerShip.Position.X;
  double dy = mstate.Y - playerShip.Position.Y;
  playerShip.Angle = (float)Math.Atan2( dy, dx );
}
```

6.6 Hiiren käyttäminen ilman kursoria

Jos hiirellä esimerkiksi ohjataan pelihahmoa tai liikutellaan kameraa pelimaailmassa, kursorin koordinaatteja hyödyllisempi tieto on se, mihin suuntaan ja kuinka

paljon hiirtä on liikutettu viime tarkistuksesta. XNA:ssa ei ole suoraa tukea tällaiselle suhteelliselle hiiren kuuntelulle, mutta se voidaan toteuttaa itse pienellä vaivalla.

Intuitiivinen tapa ratkaista ongelma olisi merkitä muistiin hiiren edelliset koordinaatit ja vähentää ne uusista koordinaateista `Update`-metodissa. Ongelmia tulee kuitenkin, kun saavutetaan näytön reuna. Vaikka kursori olisi näkymätön, sen koordinaatit on rajoitettu nollan ja käytössä olevan näyttöresoluution välille (Reed 2009, s. 46).

Ratkaisu on yksinkertainen, mutta ruma: asetetaan joka päivityskerralla hiiren kursori ruudun keskelle. Ruudun keskipiste voidaan selvittää ja tallentaa muuttujaan `Initialize`-metodissa. Samalla voidaan asettaa hiiren kursori valmiiksi keskelle ruutua, ettei ensimmäinen päivitys johda odottamattomiin äkkiliikkeisiin. (Reed 2009, s. 45-46)

```
int centerX, centerY;

protected override void Initialize()
{
  centerX = Window.ClientBounds.Width / 2;
  centerY = Window.ClientBounds.Height / 2;
  Mouse.SetPosition( centerX, centerY );
  // ...
}
```

Nyt hiiren liike saadaan `Update`-metodissa vähentämällä kursorin paikkakoordinaateista ruudun keskipisteen koordinaatit. Hiiren kursori on muistettava asettaa ruudun keskelle joka päivityksen jälkeen.

```
MouseState mstate = Mouse.GetState();
int xmuutos = mstate.X - centerX;
int ymuutos = mstate.Y - centerY;

// tehdään jotain

Mouse.SetPosition( centerX, centerY );
```

7 Pelilogiikka ja törmäykset 2d-maailmassa

Toistaiseksi pelistämme puuttuu vielä kokonaan pelattavuus: vihollisaluksia ei ole kuin yksi, ja sekin pysyttelee täysin paikallaan. Tässä luvussa muutamme tilannetta lisäämällä satunnaisesti generoituvia liikkuvia vihollisia, pelaajalle mahdollisuuden ampua ja vihollisille kyvyn ampua takaisin. Samalla opimme tekemään ajastettuja toimintoja ja satunnaisuutta, käyttämään törmäystarkistusta ja ohjelmoimaan alkeellisen tekoälyn.

7.1 Vihollinen liikkumaan tekoälyn ohjaamana

Tällä hetkellä sekä pelaajan että vihollisen alus ovat `GameObject`-luokan olioita. Pelaajan alusta liikutetaan `Update`-metodissa sen perusteella, mitä näppäimiä on painettu. Ensimmäisenä mieleen tuleva ratkaisu on liikuttaa myös vihollisia `Update`-metodissa. Yksinkertaisessa pelissä tämä lähestymistapa voi toimiakin, mutta jos tekoälytyyppejä on useampia (esimerkiksi pelaajaa seuraava, pelaajaa pakeneva ja satunnainen), tarvitaan ylimääräisiä tarkistuksia ja `Update`-metodi voi paisua satojen rivien mittaiseksi. Seuraavaksi yksinkertaisin, mutta huomattavasti siistimpi tapa on luoda kullekin vihollistyypille oma luokkansa.

Tehdään siis uusi luokka nimellä `Enemy`, ja peritään se aiemmin tekemästämme luokasta `GameObject`. Lisätään rakentaja, jossa asetamme vihollisen oletusnopeuden ja otamme peliluokan viitteen talteen, sekä `Update`-metodi, johon itse tekoäly tulee.

```
public class Enemy : GameObject
{
  Game1 game;

  public Enemy( Game1 game )
  {
    this.Game = game;
    Speed = 0.1f;
  }
  public override void Update( GameTime gameTime )
  {
    base.Update( gameTime );
  }
}
```

`Update`-metodin rivi `base.Update( gameTime )` tarkoittaa, että kutsutaan kantaluokan (`GameObject`) vastaavaa metodia. Tämä tehdään yleensä ylikirjoitetun metodin lopussa (Hejlsberg & Wiltamuth ja Golde 2006, s. 311–312).

Korjataan vielä pääpelitiedosto `Game1.cs` käyttämään uutta luokkaamme, ja lisätään samalla julkinen `Random`-olio satunnaislukujen generointiin.

```
public class Game1 : Microsoft.Xna.Framework.Game
{
  public Random Rnd = new Random();
  // ...
  GameObject playerShip;
  Enemy enemyShip;
  // ...
  protected override void LoadContent()
  {
    // ...
    enemyShip = new Enemy( this );
    // ...
  }
  // ...
```

Kun ajamme ohjelman, vihollisen pitäisi nyt lähteä suoraan eteenpäin, mikä seuraa rakentajassa asetetusta oletusnopeudesta. Nyt meillä on tarvittavat ainekset kasassa siirtyäksemme tekemään itse tekoälyä `Update`-metodiin.

Satunnaisuuden lisäksi reaaliaikaisessa pelissä tarvitsemme aikaa. Kaukaa viisaina toimme `Update`-metodiin `GameTime`-tyyppisen olion, joka pitää yllä aikaa pelin kestosta ja viimeisimmästä päivityksestä. Voimme käyttää jälkimmäistä, jos haluamme esimerkiksi vaihtaa suuntaa tietyin aikavälein.

```
double moveTimer;
float bank;

public override void Update( GameTime gameTime )
{
  moveTimer -= gameTime.ElapsedGameTime.TotalSeconds;

  if ( moveTimer < 0 )
  {
    moveTimer = 0.016;
```

```
    if ( game.Rnd.Next( 100 ) > 95 )
      bank = (float)game.Rnd.NextDouble() - 0.5f;

    if ( game.Rnd.Next( 100 ) > 60 )
    {
      Angle += 0.2f * bank * MathHelper.Pi;
    }
  }

  base.Update( gameTime );
}
```

7.2 Kääriytyvä pelialue

Nyt kun vihollisella on jonkinlainen tekoäly, se lentää hyvin helposti ulos näkyvältä pelialueelta, kuten myös pelaaja. Tähän on useita eri ratkaisuja: alus tuhoutuu jos se osuu reunaan, kamera seuraa pelaajan alusta niin että se pysyy koko ajan näytön keskellä tai pelialue *kääriytyy* eli reunan yli mentäessä ilmestytään vastakkaisen reunan takaa pelialueelle kuten kaksiulotteisen toruksen pinnalla. Tämä alaluku kuvaa yksinkertaisen tavan toteuttaa jälkimmäinen vaihtoehto.

Pelialueen kääriytyminen voidaan toteuttaa helpoiten ottamalla peliolioiden koordinaateista jakojäännös pelikentän leveyden tai pituuden suhteen. Todelliset koordinaatit ovat siis

```
X = annettuX % alueenLeveys;
Y = annettuY % alueenKorkeus;
```

Ongelmia tosin syntyy, jos annettu koordinaatti onkin negatiivinen. Sillä oletuksella, että koordinaatteja ei yritetä asettaa useamman pelikentän mitan verran negatiivisiksi, voidaan käyttää kaavoja

```
X = ( alueenLeveys + annettuX ) % alueenLeveys;
Y = ( alueenKorkeus + annettuY ) % alueenKorkeus;
```

Koodiksi muunnettaessa C#-ominaisuuksien asetusmetodi `set` (Hejlsberg & Wiltamuth ja Golde 2006, s. 319–324) osoittaa hyödyllisyytensä. Koodi tulee `GameObject`-luokkaan `Position`-ominaisuuden tilalle.

```
private Vector2 position;
```

```
private float wrapX;
private float wrapY;

public Vector2 Position
{
    get { return position; }
    set
    {
        float x = ( wrapX + value.X ) % wrapX;
        float y = ( wrapY + value.Y ) % wrapY;
        position = new Vector2( x, y );
    }
}
```

Tarvitsemme vielä tiedon pelikentän dimensioista (leveydestä ja pituudesta) sijoitettavaksi attribuutteihin `wrapX` ja `wrapY`. Tämä onnistuu helpoiten rakentajassa, jolle annetaan parametrina peliluokka.

```
public GameObject( Game game )
{
    wrapX = game.Window.ClientBounds.Width;
    wrapY = game.Window.ClientBounds.Height;
}
```

Myös `Enemy`-luokan rakentajaa täytyy muuttaa kutsumaan oikeaa kantaluokan rakentajaa ja välittämään sille tarvittavan parametrin.

```
public Enemy( Game1 game )
    : base( game )
{
    this.game = game;
    Speed = 0.05f;
}
```

Kun rakentajien kutsut vielä korjataan (peliluokassa itsessään parametrina on `this`), pelialueesta tulee kääriytyvä.

7.3 Ampuminen

Ampuminen on oleellinen osa avaruudessa (tai missä tahansa muuallakin) taistelua. Ammukset ovat peliolioita siinä missä aluksetkin, joskin niiden olemassaoloaika on yleensä lyhyt eikä tekoälyä tai pelaajan ohjausta ole. Osumien käsittelyyn palaamme myöhemmin törmäystarkistusten yhteydessä.

Kun sopiva tekstuuri on haettu tai tehty ja lisätty projektiin, voidaan siirtyä pohtimaan ammusten ominaisuuksia. Pelattavuuden ja osittain tehokkuudenkin kannalta ei ole järkevää, että pelikenttä täyttyy ammuksista. Näin käy helposti, kun pelialue on kääriytyvä ja olemassaolevat ammukset jäävät radoilleen ikuisiksi ajoiksi. Siksi on järkevää asettaa maksimielinaika, jonka ylitettyään ammus häviää kokonaan pelistä.

```
public class Projectile : GameObject
{
  public double Lifetime { get; set; }
  public double MaxLifetime { get; set; }

  public Projectile( Game game, double lifetime )
    : base( game )
  {
    MaxLifetime = lifetime;
  }

  public override void Update( GameTime gameTime )
  {
    Lifetime += gameTime.ElapsedGameTime.TotalSeconds;
    base.Update( gameTime );
  }
}
```

Ammukset on myös tallennettava johonkin tietorakenteeseen, mistä `Update`- ja `Draw`-metodit pääsevät niihin käsiksi. Lista sopii hyvin tarkoitukseen.

```
List<Projectile> projectiles = new List<Projectile>();
```

`Projectile`-olioiden `Update`-metodin kutsumisen lisäksi pelin `Update`-metodissa on syytä tarkistaa, onko ammuksilla vielä elinaikaa jäljellä. ”Vanhentuneet” ammukset voidaan poistaa listasta, sillä niille ei ole enää mitään käyttöä. Pelin `Draw`-metodissa on myös kutsuttava olion `Draw`-metodia, että ammukset piirretään ruu-

dulle.

```
List<Projectile> projectiles = new List<Projectile>();
// ...
protected override void Update( GameTime gameTime )
{
  // ...
  for ( int i = projectiles.Count - 1; i >= 0; i-- )
  {
    projectiles[i].Update( gameTime );

    if ( projectiles[i].Lifetime >
      projectiles[i].MaxLifetime )
        projectiles.RemoveAt( i );
  }
}
```

```
protected override void Draw( GameTime gameTime )
{
  GraphicsDevice.Clear( Color.Black );
  spriteBatch.Begin();

  // ...

  for ( int i = 0; i < projectiles.Count; i++ )
  {
    projectiles[i].Draw( spriteBatch );
  }

  spriteBatch.End();
  base.Draw( gameTime );
}
```

Jälkimmäisessä metodissa voidaan käyttää myös `foreach`-silmukkaa, joskin silloin täytyy varautua poikkeukseen, joka aiheutuu jos listan sisältö muuttuu piirron aikana (Hejlsberg & Wiltamuth ja Golde 2006, s. 246–248).

Näiden valmistelujen jälkeen voimme siirtyä itse ampumiseen. Yksinkertaisuuden vuoksi toteutamme ampumismetodin julkisena peliluokkaan. Parempi tapa

olisi tehdä alukselle oma luokkansa (`GameObject`-luokasta perittynä) ja lisätä ampumismetodi sinne. Ammusluokkaan olisi myös lisättävä staattinen ominaisuus tekstuurille, joka alustettaisiin `LoadContent`-metodissa.

```
public void Shoot( GameObject shooter )
{
  Projectile p = new Projectile( this, 0.5f );
  p.Texture = Content.Load<Texture2D>( "ammo" );
  p.Position = shooter.Position;
  p.Angle = shooter.Angle;
  p.Speed = 0.3f;
  projectiles.Add( p );
}
```

Pelaajan ampuminen toteutetaan kuten liikkuminenkin, eli kuuntelemalla näppäinten painalluksia.

```
if ( kbstate.IsKeyDown( Keys.Space ) &&
     oldKbState.IsKeyUp( Keys.Space ) )
{
  Shoot( playerShip );
}
```

Vihollisalukset pääsevät `Shoot`-metodiin käsiksi pelin olioviitteen `game` kautta. Ampumisen lisäksi vihollisen olisi hyvä tietää myös pelaajan paikka, joten myös `playerShip` kannattaa tehdä julkiseksi attribuutiksi tai ominaisuudeksi. Vihollisen ampumiskoodi voi olla vaikka tällainen:

```
Vector2 playerDist = game.playerShip.Position - Position;
double distAngle = Math.Atan2( playerDist.Y, playerDist.X );

if ( Math.Abs( Angle - distAngle ) < Math.PI / 4 )
  game.Shoot( this );
```

Nyt vihollinen ampuu sarjatulta aina kun pelaaja on sen eteen muodostaman 45 asteen keilan sisällä.

7.4 Vihollisten generointi

Tähän asti kentällä on ollut vain yksi vihollinen, jolle on riittänyt yksittäinen `Enemy`-tyyppinen attribuutti peliluokassa. Jos haluamme vaihtelevan määrän satunnais-

esti generoituvia vihollisia, on järkevää tehdä vihollisista lista. XNA:ssa itsessään ei ole tapahtumapohjaista ajastinluokkaa, jota voisi käskeä luomaan uusi vihollinen satunnaiseen paikkaan n sekunnin välein, mutta sellainen kannattaa tehdä itse jos pelissä on enemmälti ajastettua toiminnallisuutta. Tässä esimerkissä tyydymme kuitenkin käyttämään `Update`-metodin `gameTime`-parametria ja joukkoa peliluokan attribuutteja.

```
List<Enemy> enemies = new List<Enemy>();
const int maxEnemies = 10;
const double spawnTime = 5;
double spawnTimer;
```

Piirto- ja päivityskoodi on hyvin samantyyppinen kuin ammuksien vastaava, lukuunottamatta eliniän tarkistusta. Toisin sanoen `enemies`-lista käydään läpi `for` (tai `foreach`) -silmukassa `Update` ja `Draw` -metodeissa ja kutsutaan vastaavaa `Enemy`-luokan metodia. Kaikki `enemyShip`-olioon viittaava koodi voidaan poistaa tarpeettomana.

Itse generointikoodi tulee peliluokan `Update`-metodiin. Koodista kannattaa tehdä oma metodinsa esimerkiksi nimellä `SpawnEnemies`, kuten myös kontrollien kuuntelusta, jos sitä ei ole tehnyt jo. Näin metodeista ei tule turhan pitkiä, mikä helpottaa mahdollisten virheiden etsintää.

```
protected override void Update( GameTime gameTime )
{
  UpdateControls();
  UpdateObjects( gameTime );
  SpawnEnemies( gameTime );
  base.Update( gameTime );
}

protected override void Draw( GameTime gameTime )
{
  GraphicsDevice.Clear( Color.Black );
  spriteBatch.Begin();

  for ( int i = 0; i < projectiles.Count; i++ )
  {
    projectiles[i].Draw( spriteBatch );
  }
```

```
  for ( int i = 0; i < enemies.Count; i++ )
  {
    enemies[i].Draw( spriteBatch );
  }

  spriteBatch.End();
  base.Draw( gameTime );
}

void SpawnEnemies( GameTime gameTime )
{
  if ( enemies.Count >= maxEnemies )
    return;

  spawnTimer += gameTime.ElapsedGameTime.TotalSeconds;

  if ( spawnTimer < spawnTime )
    return;

  spawnTimer = 0;

  Enemy newEnemy = new Enemy( this );
  double x = Window.ClientBounds.Width * Rnd.NextDouble();
  double y = Window.ClientBounds.Height * Rnd.NextDouble();
  double a = 2 * Math.PI * Rnd.NextDouble();

  newEnemy.Position = new Vector2( (float)x, (float)y );
  newEnemy.Angle = (float)a;
  newEnemy.Texture = shipTex;
  enemies.Add( newEnemy );
}
```

7.5 Tietorakenteiden optimointia törmäystarkastuksien helpottamiseksi

Tähän mennessä olemme tallentaneet peliolioita useampiin tietorakenteisiin. Peliluokastamme löytyy pelaajan aluksen lisäksi lista vihollisista ja toinen lista ammuk-

sista. Näiden yhdistäminen yhdeksi listaksi säästää muistin lisäksi myös koodirivejä etenkin `Update`- ja `Draw` -metodeissa.

Poistetaan sekä ammus- että vihollislista attribuuteista ja tehdään tilalle uusi lista `Objects`, jonka alkiot ovat tyyppiä `GameObject`. Pelaajan alus `playerShip` on syytä jättää omaksi attribuutikseen, sillä sitä tarvitaan vihollisaluksen tekoälyn kohdistusta varten. Se kannattaa silti lisätä listaan, niin sitä ei tarvitse käsitellä erityistapauksena piirron ja päivityksen yhteydessä.

Yhteensopivuuden lisäämiseksi ja olioiden poistamisen helpottamiseksi voidaan lisätä `GameObject`-luokkaan boolean-tyyppinen ominaisuus `Alive`, joka kertoo, onko peliolio elossa. `Projectile`-luokka voi ylikirjoittaa ominaisuuden tarkistamaan, onko elinaikaa vielä jäljellä.

```
public class GameObject
{
  public virtual bool Alive { get; set; }

  public GameObject( Game game )
  {
    Alive = true;
    wrapX = game.Window.ClientBounds.Width;
    wrapY = game.Window.ClientBounds.Height;
  }
```

```
public class Projectile
{
  public override bool Alive
  {
    get { return base.Alive && Lifetime < MaxLifetime; }
    set { base.Alive = value; }
  }
  // ...
```

`Update`-metodista omaksi metodikseen irrotettu `UpdateObjects` voidaan muokata nyt näyttämään tältä:

```
void UpdateObjects( GameTime gameTime )
{
    for ( int i = Objects.Count - 1; i >= 0; i-- )
    {
```

```
            Objects[i].Update( gameTime );

            if ( !Objects[i].Alive )
                Objects.RemoveAt( i );
        }
    }
```

Huomaa, kuinka silmukka laskee olioita ylhäältä alaspäin. Näin olion poisto listasta käy yhdessä silmukassa indeksimuuttujaan `i` koskematta.

Myös `Draw`-metodi kutistuu kuuteen riviin aikaisemmasta yhdeksästä. Yksi silmukka putoaa pois, mutta vihollisten määrälle on tehtävä oma attribuuttinsa, sillä määrää ei voida tarkistaa enää listan koosta.

7.6 Törmäystarkistukset

Törmäystarkistukset ovat oleellinen osa mitä tahansa videopeliä, pois lukien lautaja korttipelikäännökset. Koska peliolioita on yleensä paljon, ja törmäykset täytyy tarkistaa niiden kaikkien välillä, törmäystarkistuksista tulee tasapainoilua tarkkuuden ja nopeuden välillä (Reed 2009, s. 51). Tässä alaluvussa esitellään kaksi tapaa toteuttaa törmäyksienkäsittely 2d-pelissä ja verrataan niitä lyhyesti.

Törmäystarkistuksen runko muodostuu kahdesta sisäkkäisestä silmukasta, joissa kutakin spriteparia vertaillaan keskenään. Vertailutapoja on useampia, mutta runko on sama. Tehdään se ensin `Update`-metodista kutsuttavaksi aliohjelmaksi.

```
void CheckCollisions()
{
  for ( int i = 0; i < Objects.Count; i++ )
  {
    for ( int j = i + 1; j < Objects.Count; j++ )
    {
      // Tarkistus tulee tähän
    }
  }
}
```

Kahden peliolion törmäystarkistus tapahtuu ympäröimällä niiden spritet sopivalla yksinkertaisella muodolla ja testaamalla, leikkaavatko nämä muodot toisensa. Suorakulmio (*bounding rectangle*) ja ympyrä (*bounding circle*) ovat ilmeisiä vaihtoehtoja. Yleensä valitaan se, joka tarkemmin vastaa peliolion todellista muotoa.

Tarkistuksia varten `GameObject`-luokkaan on lisättävä tiettyjä asiaankuuluvia ominaisuuksia. Suorakulmiotarkistusta varten voidaan käyttää `Rectangle`-tyyppistä ominaisuutta, jolla on valmis `Intersects`-metodi leikkauksen laskemiseen. Ympyrätarkistukselle riittää paikan lisäksi pelkkä säde. Ominaisuudet asetetaan päivittymään automaattisesti joka kerta, kun niihin vaikuttavia arvoja muutetaan.

```
public class GameObject
{
  private Vector2 position;

  // ...
  public Rectangle BoundingRect { get; set; }
  public float Radius { get; set; }

  public Vector2 Position
  {
    get { return position; }
    set
    {
      float x = ( wrapX + value.X ) % wrapX;
      float y = ( wrapY + value.Y ) % wrapY;
      position = new Vector2( x, y );
      BoundingRect = new Rectangle(
         (int)x, (int)y,
         BoundingRect.Width, BoundingRect.Height );
    }
  }

  public Texture2D Texture
  {
    get { return texture; }
    set
    {
      texture = value;
      BoundingRect = new Rectangle(
        (int)Position.X, (int)Position.Y,
        texture.Width, texture.Height );
      Radius = Math.Max( texture.Width, texture.Height );
```

```
    }
  }
  // ...
```

Suorakulmioiden leikkaukset voidaan tarkistaa valmiilla `Intersects`-metodilla:

```
if ( Objects[i].BoundingRect.Intersects(
       Objects[j].BoundingRect ) )
{
  // ...
}
```

Ympyräleikkausten tarkistaminen vaatii hieman enemmän matematiikkaa. Ympyrät leikkaavat, kun niiden keskipisteiden etäisyyksien neliöiden summa on pienempi kuin niiden yhteenlaskettujen säteiden neliö.

```
float xdist = Objects[j].Position.X - Objects[i].Position.X;
float ydist = Objects[j].Position.Y - Objects[i].Position.Y;
float rsum = Objects[i].Radius + Objects[j].Radius;

if ( xdist * xdist + ydist * ydist < rsum * rsum )
{
  // ...
}
```

Kun törmäysehto toteutuu, voidaan esimerkiksi hävittää molemmat oliot ja lisätä törmäyskohtaan komea räjähdysefekti. Siten toisiinsa tai ammuksiin osuvat alukset (ja jopa toisiinsa osuvat ammukset) näyttäisivät tuhoutuvan asianmukaisesti. Räjähdyksen tekemiseen tarvitsisimme kuitenkin animaatiotekstuuria, jollaista ei XNA:ssa vakiona ole, joten joudumme jättämään sen pois.

```
if ( xdist * xdist + ydist * ydist < rsum * rsum )
{
    Objects[i].Alive = false;
    Objects[j].Alive = false;
}
```

Nyt törmäykset alusten välillä toimivat kuten pitääkin. Sen sijaan jos alus yrittää ampua, tulee ongelmia. Alus tuhoutuu itse, mikä johtuu ammuksen sijoittamisesta samaan paikkaan ampujansa kanssa. Tähän on useita ratkaisuja: ammus voidaan

sijoittaa tarpeeksi kauaksi ampuvasta aluksesta, se voidaan asettaa virittymään vasta tietyn ajan päästä tai yksinkertaisesti asettaa tunnistamaan ampujansa ja olla osumatta siihen missään tapauksessa.

Ensimmäisen ratkaisun suurin ongelma on alusten liike. Ampuvan aluksen senhetkinen nopeus tulee ottaa oikealla tavalla huomioon tai muuten ongelma toistuu tai panos näyttää ilmestyvän tyhjästä avaruudesta aluksen eteen. Toinen ratkaisu on käytännössä sama, mutta paikan sijaan lasketaan aikaa. Kolmannessa tavassa selvitään ilman nopeuslaskuja, mutta samalla menetetään mahdollisuus osua itseensä. Tässä tapauksessa se ei ole ongelma, kun ammuksen elinikä on vähemmän kuin täyden ruudullisen verran kummassakin suunnassa.

Tunnistaakseen ampujansa, ammuksella on oltava ominaisuus sitä varten. Olkoon se nimeltään `Owner` ja tyyppiä `GameObject`. Luonteva paikka asettaa arvo on peliluokan `Shoot`-metodi.

Ennen törmäystarkistusta tulee tarkistaa, onko jompi kumpi olioista toisen ampuja. Tyyppimuunnostensa vuoksi tarkistuksessa on sen verran ohjelmakoodia, että siitä kannattaa tehdä oma aliohjelmansa.

```
void CheckCollisions()
{
  for ( int i = 0; i < Objects.Count; i++ )
  {
    for ( int j = i + 1; j < Objects.Count; j++ )
    {
      if ( IsOwnerOf( Objects[i], Objects[j] ) )
        continue;

      // Tarkistus tulee tähän
    }
  }
}
```

```
bool IsOwnerOf( GameObject s1, GameObject s2 )
{
  Projectile p1 = s1 as Projectile;
  Projectile p2 = s2 as Projectile;

  if ( p1 != null )
    return ( p1.Owner == s2 );
```

```
  if ( p2 != null )
    return ( p2.Owner == s1 );

  return false;
}
```

Suuremmilla pelialueilla ja spritemäärillä on järkevää tehdä useita sisäkkäisiä törmäystarkistuksia. Esimerkiksi pelialue voidaan jakaa sopivan kokoisiin vyöhykkeisiin, jolloin ensimmäinen tarkistus on katsoa, ovatko potentiaaliset törmääjät samalla tai naapurivyöhykkeellä. Jos ovat, voidaan tarkistusta jatkaa esimerkiksi pallomuotoon ja siitä alaspäin haluttuun tarkkuuteen. Oikein toteutettuna suurin osa törmäystarkistuksista päättyy jo ensimmäiseen tai toiseen testiin. (Reed 2009, s. 55)

8 Äänet ja musiikki

Nyt kun pelissämme on grafiikka ja pelattavuus kunnossa, on aika lisätä ääniefektit ja taustamusiikki. XNA tarjoaa kaksi eri tapaa ääniefektien toistamiseen: äänitiedostoja suoraan käyttämällä tai XNA:n mukana tulevaa XACT-työkalua (*Microsoft Cross-Platform Audio Creation Tool*) käyttämällä (Carter 2009, s. 123). Käymme läpi molemmat tavat vertailun vuoksi.

8.1 Ääniefektit yksinkertaisella tavalla

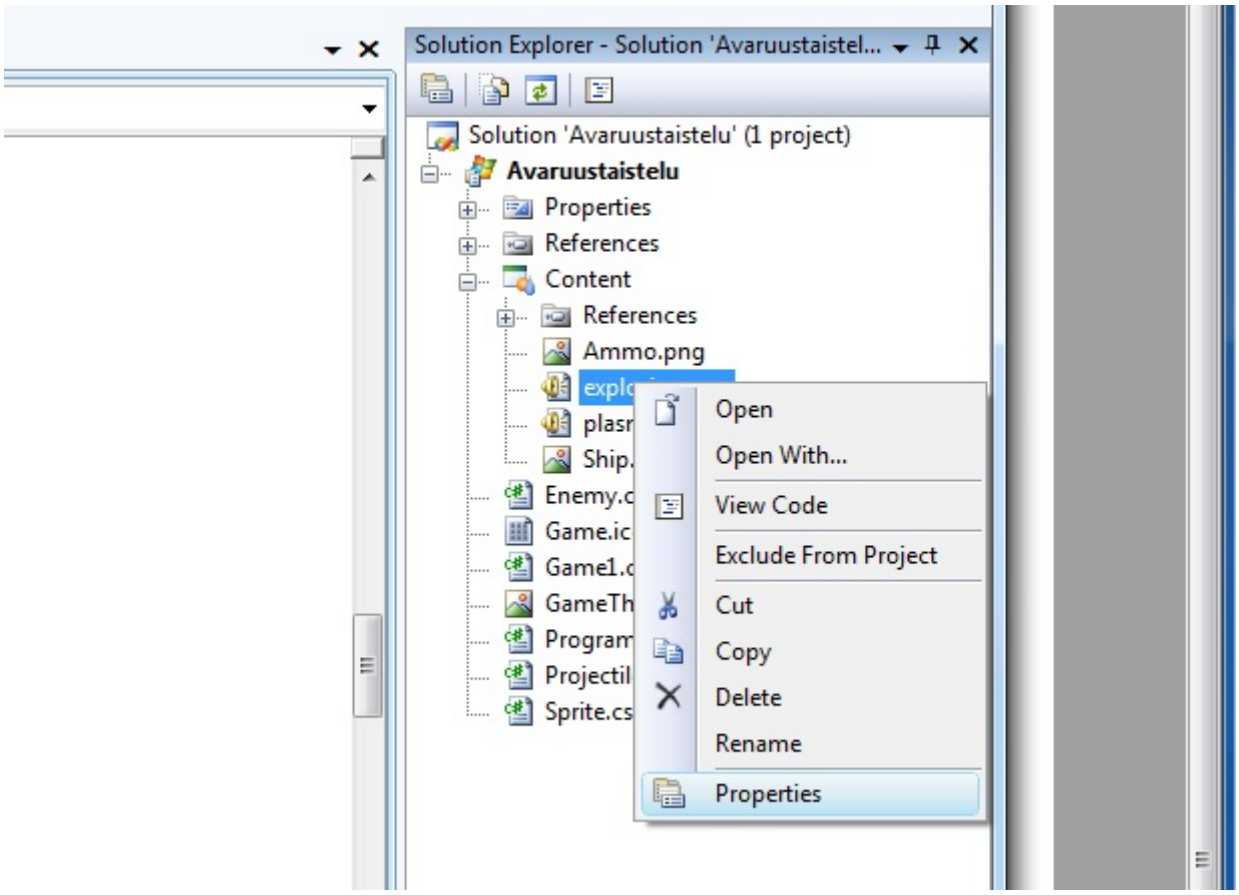
Yksinkertaisemmalla tavalla ääniefektien lisääminen projektiin tehdään samalla tavalla kuin esimerkiksi tekstuureille (Reed 2009, s. 85). Äänitiedosto lisätään projektiin, jonka jälkeen se voidaan ladata `SoundEffect`-tyyppiseen muuttujaan `LoadContent`-metodissa (Reed 2009, s. 86). Lisätään projektiimme kaksi ääniefektiä: toista käytetään kun alus ampuu, ja toista kun se tuhoutuu.

```
SoundEffect explosionSound;
SoundEffect shootSound;

protected override void LoadContent()
{
  // ...
  explosionSound = Content.Load<SoundEffect>( "explosion" );
  shootSound = Content.Load<SoundEffect>( "plasma" );
```

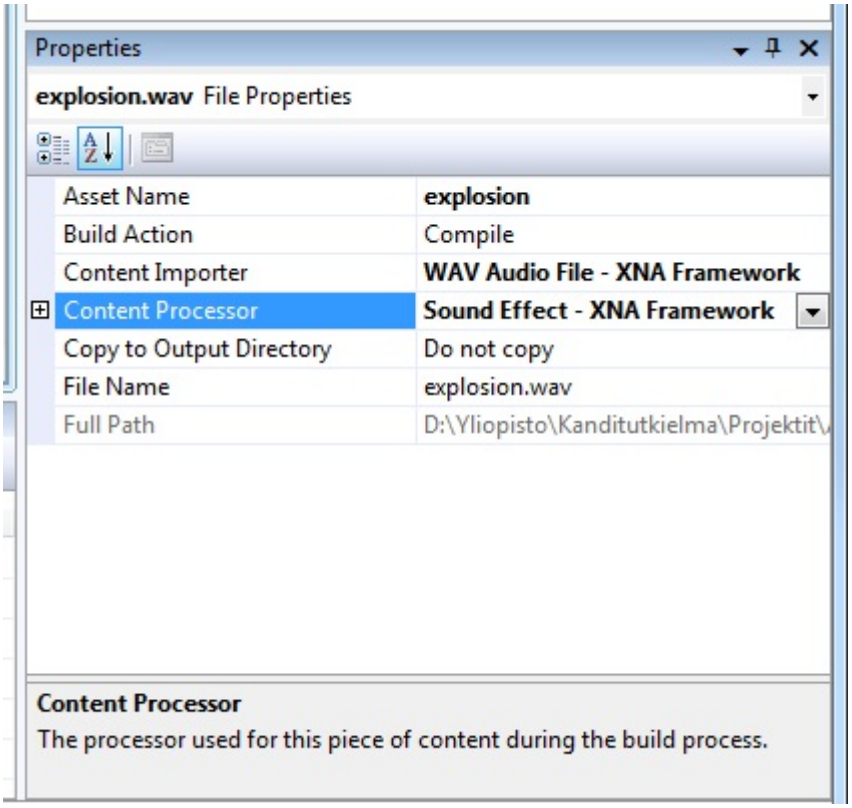
```
}
```

On hyvä tarkistaa, varsinkin mp3-tyyppisiä efektejä käytettäessä, että XNA käyttää niitä ääniefekteinä musiikin sijasta. Tämän näkee projektiin lisätyn ääniefektin ominaisuuksista, jotka löytyvät seuraavan kuvan mukaisesta paikasta.



Kuva 11: Content-olion ominaisuudet (kontekstivalikko).

Avautuvassa ikkunassa tulisi **Content Processor** -kohdassa lukea **Sound Effect - XNA Framework** ja **Content Importer** -kohdassa tiedoston tyyppi. Jos näin ei ole, seuraa käännösvirhe tai ajonaikainen poikkeus.



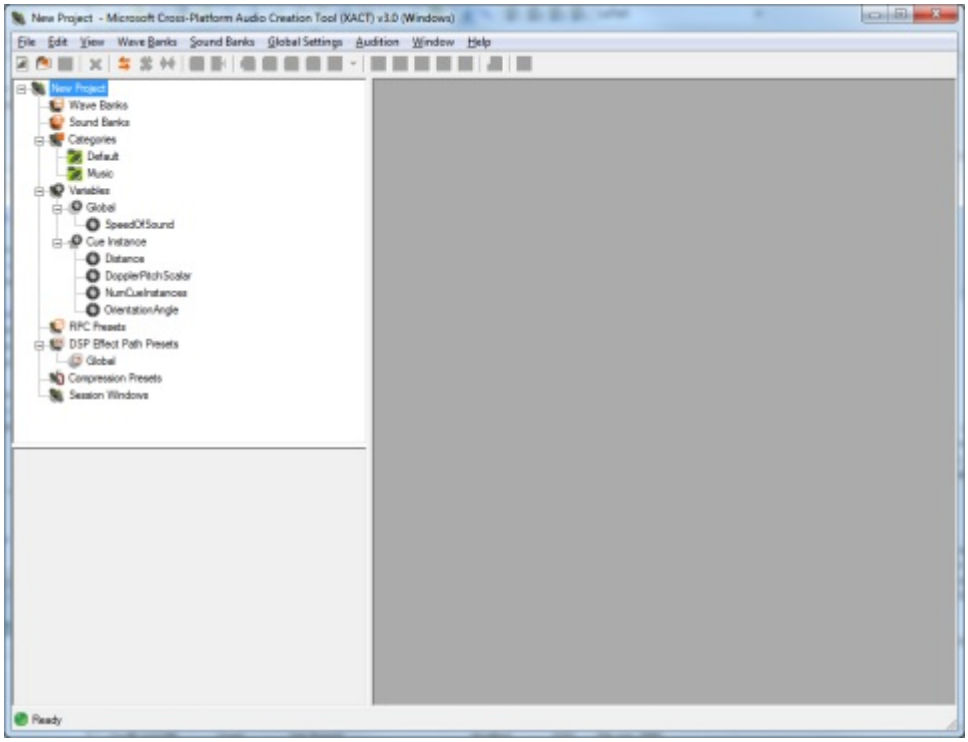
Kuva 12: Content-olion ominaisuudet (ikkuna).

Nyt kun äänet on saatu muuttujiin asti, niiden soittamiseen riittää yksi rivi: `explosionSound.Play()` tai `shootSound.Play()` (Grootjans 2009, s. 584). Metodista `Play` on myös ylikuormitettu versio, jolle voidaan antaa parametreiksi muutokset äänen voimakkuuteen, taajuuteen ja balanssiin (Grootjans 2009, s. 584). Esimerkiksi kutsu `shootSound.Play( 0.5f, 0.3f, -1.0f )` soittaa räjähdysäänen puolella voimakkuudella nopeutettuna kokonaan vasemman kaiuttimen kautta.

8.2 Ääniefektit XACT:in kautta

Kun ääniefektejä on kymmeniä tai enemmän, niiden lataaminen koodissa yksittäin alkaa näyttää rumalta. Tähän ratkaisuksi Microsoft on kehittänyt XACT-työkalun, jolla ääniefektit voidaan koota äänipankeiksi. XACT:ia käyttämällä on myös mahdollista yhdistellä ääniefektejä, säädellä äänen laatua ja tehdä toistuvia efektejä ilman koodia tai useita äänitiedostoja, jotka sisältävät saman efektin vain hiukan muunneltuna. XACT-tuki löytyy sekä Windowsista että Xbox 360:sta, mutta ei Zunesta. (Reed 2009, s. 76–77)

XACT-työkalu löytyy käynnistysvalikon *Microsoft XNA Game Studio 3.1* -kansion alikansiosta *Tools* nimellä *Microsoft Cross-Platform Audio Creation Tool 3 (XACT3)*.



Kuva 13: Xact 3 -ohjelma.

Käyttöliittymältään XACT on suhteellisen epäintuitiivinen, eikä *Help*-valikossa yksinään seisova *About* juuri käyttöä helpota. Microsoftin ohjesivuilta (MSDN (2009)) löytyvillä tutoriaaleilla pääsee alkuun, mutta peruskäyttöä pidemmälle mentäessä hyvä XNA-kirja (esim. Carter (2009)) tai ohjelmointisivusto on lähes pakollinen.

Ensimmäinen askel XACT:in käyttöönotossa on uuden projektin luominen (*File->New Project*). XACT kysyy luotavan projektitiedoston nimeä ja sijaintia, jonka jälkeen pääsemme muokkaamaan itse projektia.

Äänitiedostojen lisääminen projektiin onnistuu luomalla niille erillinen *aaltopankki* (eng. *wave bank*). Yksi projekti voi sisältää useita aaltopankkeja, mutta jos äänitiedostoja ei ole satoja, yksi yleensä riittää. Äänitiedostot lisätään listaan kontekstivalikon *Insert Wave Files...* -kohdasta.

Aaltopankki määrittää kuitenkin vain *mitä* toistetaan, sen *miten* toistetaan säätää *äänipankki* (eng. *sound bank*). Äänipankki sisältää *ääniä* (eng. *sounds*) ja *toistovihjeitä* (eng. *cues*). Yhdestä äänitiedostosta on mahdollista luoda useampia ääniä raahaamalla ja pudottamalla niitä aaltopankki-ikkunasta äänipankki-ikkunaan. Äänen ominaisuuksista voi säädellä esimerkiksi sen voimakkuutta ja korkeutta. Toistovihjeet

puolestaan koostuvat yhdestä tai useammasta äänestä, joita voi asettaa toistumaan joko satunnaisesti tai järjestyksessä. (Grootjans 2009, s. 587).

Äänitiedostoja, ääniä ja toistovihjeitä voi kuunnella space-näppäimellä, jos kuunteluohjelma (XACT Auditioning Utility) on taustalla käynnissä. Kuunteluohjelma tosin kaikessa viisaudessaan käyttää TCP-porttia 80, joten ohjelman käyttö samalla tietokoneella web-serverin kanssa ei tule kyseeseen. (Carter 2009, s. 124).

Kun projektissa on ainakin yksi toimiva toistovihje, voidaan se tallentaa ja siirtyä Visual Studion puolelle. XACT-projekti lisätään peliprojektiin tuttuun tapaan (*Content->Add Existing Item*), jolloin Visual Studio käännöstä tehdessään kääntää yhden .xgs-tarkenteisen binääritiedoston XACT-projektille, sekä .xwb-tiedoston jokaiselle aalto- ja .xsb-tiedoston jokaiselle äänipankille. (Grootjans 2009, s. 588).

Jokaista tiedostoa kohti peliin tarvitaan olio. Nämä oliot alustetaan seuraavasti:

```
AudioEngine audioEngine;
WaveBank waveBank;
SoundBank soundBank;

protected override void Initialize()
{
  audioEngine = new AudioEngine(
    "Content/AvaruusTaistelu.xgs" );
  waveBank = new WaveBank( audioEngine,
    "Content/Waves.xwb" );
  soundBank = new SoundBank( audioEngine,
    "Content/Sounds.xsb" );
  // ...
}
```

Näistä `SoundBank`-tyyppistä oliota voidaan käskeä soittamaan toistovihjeitä.

```
soundBank.PlayCue( "explosion" );
```

Jotta toistovihjeet eivät jäisi turhaan muistiin, on `AudioEngine`-luokkaa päivitettävä tasaisin väliajoin pelin `Update`-metodissa. (Grootjans 2009, s. 588–589).

```
protected override void Update( GameTime gameTime )
{
  // ...
  audioEngine.Update();
}
```

8.3 Taustamusiikki

Taustamusiikkia voidaan soittaa periaatteessa kuten muitakin ääniä, mutta XNA tarjoaa sitä varten erikoistetut luokkansa. Mp3- ja wma-muotoiset tiedostot tunnistuvat projektiin lisättäessä automaattisesti muotoon `Song` (ks. ensimmäinen alaluku), ja niitä voidaan toistaa `MediaPlayer`-luokan staattisilla metodeilla. (Miles 2009, s. 125–126).

```
Song bgMusic;

protected override void LoadContent()
{
  bgMusic = Content.Load<Song>( "taustamusiikki" );
  // ...
}

protected override void Initialize()
{
  MediaPlayer.Play( bgMusic );
  // ...
}
```

XNA 4.0 tarjoaa mahdollisuuden toistaa kappaleita myös verkosta. (MSDN 2009)

```
Uri uriStreaming = new Uri( "http://..." )
Song bgMusic = Song.FromUri( "kappale", uriStreaming );
MediaPlayer.Play( bgMusic );
```

9 3d-grafiikka

Tähän asti käsitellyt grafiikkaesimerkit ovat kaikki olleet kaksiulotteisia. Tässä luvussa tarkastelemme kolmiulotteisen grafiikan piirtämistä XNA:n tarjoamilla luokilla ja rutiineilla luomalla ensin kameran ja katsomalla 3d-malleja sen lävitse. Kameran liikuttamista lukuunottamatta varsinainen pelilogiikka sivuutetaan, sillä huomattavaa eroa 2d-pelien logiikkaan kolmannen koordinaattiakselin lisäksi ei juurikaan ole.

9.1 Kamera komponenttina

Koska näyttölaitteen kuvapinta on kaksiulotteinen, kolmiulotteisen pelimaailman kuvaaminen sille tuottaa ongelman: miten pudottaa yksi ulottuvuus pois niin, että tilan vaikutelma ja etäisyyksien suhteet säilyvät? Lineaarialgebran vastaus ongelmaan on projektio. Samaan tapaan kuin filmi- tai digitaalikameralla voidaan ottaa kuva fyysisestä maailmastamme, voidaan pelimaailmaan sijoittaa kamera, joka kuvaa alueen kolmiulotteisesta pelimaailmasta pelaajan näyttöruudulle. XNA:ssa ei ole valmista luokkaa kameralle, sillä jokainen peli vaatii erilaisen kameran. Avaruuslentelypelissä kamera liikkuu aluksen mukana kaikkiin kolmeen ulottuvuuteen, kun taas rooli- tai strategiapelissä vapaasti liikuteltava kamera katsoo yksiköitä ylhäältäpäin (Grootjans 2009, s. 30–76).

Jos pelissä käytetään vain yhtä kameraa (kuten tavallista), kannattaa harkita sen toteuttamista komponenttina (ks. luku 4.3), mikä säästää ohjelmakoodia `Update`- ja `Draw`-metodeissa.

9.2 Kameran vektorit ja matriisit

Kameran sijainti ja kierto maailmassa voidaan määritellä kolmen vektorin avulla. Sijainnin eli paikkavektorin lisäksi määritellään suunta johon kamera osoittaa, sekä suunta joka kamerasta katsottuna on ylöspäin. Nämä kaksi jälkimmäistä vektoria ovat aina kohtisuorassa toisiaan vasten (Grootjans 2009, s. 588–589). Suunta- ja ylösvektoreiden pituuksilla ei varsinaisesti ole väliä, mutta myöhempien laskutoimitusten helpottamiseksi vektorit on hyvä normeerata, eli pitää yhden yksikön mittaisina (Grootjans 2009, s. 232).

```
public Vector3 Position { get; set; }
public Vector3 Forward { get; private set; }
public Vector3 Up { get; private set; }
```

Avainsana `private` tekee ominaisuuksista `Forward` ja `Up` muutettavia vain luokan sisällä (Hejlsberg & Wiltamuth ja Golde 2006, s. 19). Tämä on tärkeää, sillä emme halua kenenkään sotkevan vektoreillemme laskettuja arvoja.

Jotta kameraa voitaisiin käyttää kuvaamaan 3d-maailma 2d-tasoksi, tarvitaan muunnosmatriisi. Tätä matriisia kutsutaan *projektiomatriisiksi* (engl. *projection matrix*). XNA sisältää valmiin luokan `Matrix` 4x4-kokoisten muunnosmatriisien käsittelyyn, josta löytyy myös metodi `CreatePerspectiveFieldOfView` projektiomatriisin luomiseen. Metodi ottaa parametreikseen näkökentän leveyden radiaaneina, kuvasuhteen sekä lähi- ja kaukoleikkaustason etäisyyden. Näistä parametreista

muodostuu katkaistun nelikulmiopohjaisen kartion muotoinen "keila"(eng. *viewing frustum*), jonka sisällä olevat kappaleet kamera näkee. Kaikki, mikä on näkökentän ulkopuolella, lähileikkaustason edessä tai kaukoleikkaustason takana, jätetään piirtämättä. (Reed 2009, s. 184–187)

Tarvitsemme myös toisen matriisin kameralle, *näkymämatriisin* (eng. *view matrix*). Näkymämatriisi sisältää periaatteessa samat tiedot kuin vektorit `Position`, `Forward` ja `Up`, mutta tulemme tarvitsemaan matriisimuotoa muunnoksiin.

Projektio- ja näkymämatriiseja tarvitaan myöhemmin piirtämisessä, joten niistä on syytä tehdä julkisesti luettavia.

```
public Matrix Projection { get; private set; }
public Matrix View { get; private set; }
```

Matriisit voidaan alustaa kameraluokan `Initialize`-metodissa.

```
public override void Initialize()
{
  // Build camera projection matrix
  Projection = Matrix.CreatePerspectiveFieldOfView(
    MathHelper.PiOver4,
    Game.GraphicsDevice.Viewport.AspectRatio,
    1, 3000 );

  // Build camera view matrix
  UpdateView();

  base.Initialize();
}
```

Projektiomatriisi pysyy samana koko pelin ajan, eli tässä tapauksessa kuvaa 45 asteen keilan, joka leikkautuu lähellä yhden ja kaukana kolmentuhannen yksikön päässä kamerasta. Kuvasuhde otetaan suoraan ikkunan tai näyttöruudun (koko ruudun tilassa) kuvasuhteesta. Näkymämatriisi sen sijaan on päivitettävä joka kerta kun kameraa siirretään tai käännellään, siksi erillinen metodi `UpdateView` (Grootjans 2009, s. 46–47). XNA:n `Matrix`-luokasta löytyy metodi myös näkymämatriisin luomiseen.

```
private void UpdateView()
{
  Vector3 origTarget = new Vector3( 0, 0, -1 );
```

```
  Vector3 origUp = new Vector3( 0, 1, 0 );

  View = Matrix.CreateLookAt( Position, origTarget, origUp );
}
```

Tulemme muokkaamaan `UpdateView`-metodia vielä myöhemmin, kun teemme kamerasta pyöritettävän. Jotta näkymämatriisi pysyisi ajan tasalla, sen päivittäminen `Update`-metodissa on tarpeen.

```
public override void Update( GameTime gameTime )
{
  UpdateView();
  base.Update( gameTime );
}
```

9.3 3d-mallit ja pelioliot

Jotta saisimme ruudulle varsinaista 3d-grafiikkaa muuten kuin piste (verteksi) kerrallaan, tarvitsemme 3d-malleja. Mallit lisätään projektiin samaan tapaan kuin muukin sisältö (ks. luku 4.4).

Tehdään pelioliolle samaan tapaan oma luokka kuin aiemmassa kaksiulotteisessa peliesimerkissämme. Mallin lisäksi toinen oleellinen 3d-peliolion ominaisuus on sen *maailmanmatriisi* (eng. *world matrix*), johon sisältyy tieto sen paikasta, koosta ja kierrosta (Reed 2009, s. 195).

```
public class GameObject3d
{
    public Model Model { get; set; }
    public Matrix World { get; set; }
}
```

Maailmanmatriisia voidaan muokata `Matrix`-luokan metodeilla `CreateTranslation` (translaatio eli paikan muunnos), `CreateRotation?` (rotaatio eli kierto, missä ? on akseli jonka ympäri kierretään) ja `CreateScale` (skaalaus eli koon muunnos) (Carter 2009, s. 77–80). Näistä metodeista saatavat arvot voidaan joko sijoittaa suoraan maailmanmatriisiin, tai kertoa keskenään muunnosten yhdistämiseksi (Carter 2009, s. 57).

Esimerkkinä peliolion alkupaikan määritys rakentajassa:

```
public GameObject3d( Vector3 startingPos )
```

```
{
  World = Matrix.CreateTranslation( startingPos );
}
```

Lopuksi tarvitsemme vielä metodin, joka piirtää kappaleen. Koska eri kappaleet tai niiden osat voidaan piirtää eri tavoilla esimerkiksi niiden valaistuksesta ja pintamateriaalista riippuen, ei yleispätevää `Model.Draw` -kutsua ole. Sen sijaan mallit piirretään osa (*mesh*) kerrallaan käyttäen ns. *efektejä*, jotka määrittelevät, miten piirtäminen tehdään.

Efektien tekemiseen XNA tarjoaa erillisen HLSL-varjostinohjelmointikielen, jonka syntaksi muistuttaa läheisesti C-kieltä. Efektit koostuvat *verteksi-* ja *pikselivarjostimista*, joista ensimmäisiä käytetään muokkaamaan ruudulla näkyviä verteksejä eli näkymäkartion sisällä olevia kolmiulotteisia pisteitä. Jälkimmäiset muuttavat projektion jälkeen ruudulla näkyviä kaksiulotteisia pisteitä eli pikseleitä. Kaikki kolmiulotteinen piirtäminen XNA:ssa tapahtuu efektejä käyttäen. Yksinkertaisuuden vuoksi kuitenkin sivuutamme HLSL-ohjelmoinnin ja käytämme XNA:n tarjoamaa valmista efektiä `BasicEffect`. (Reed 2009, s. 275–277).

```
public void Draw( Camera cam )
{
  Matrix[] transforms = new Matrix[Model.Bones.Count];
  Model.CopyAbsoluteBoneTransformsTo( transforms );

  foreach ( ModelMesh mesh in Model.Meshes )
  {
    foreach ( BasicEffect be in mesh.Effects )
    {
      be.EnableDefaultLighting();
      be.Projection = cam.Projection;
      be.View = cam.View;
      be.World = World;
    }

    mesh.Draw();
  }
}
```

`GameObject3d`-luokkamme on nyt valmis. Lisätään se peliin attribuutiksi, ladataan sille 3d-malli ja lopuksi piirretään se.

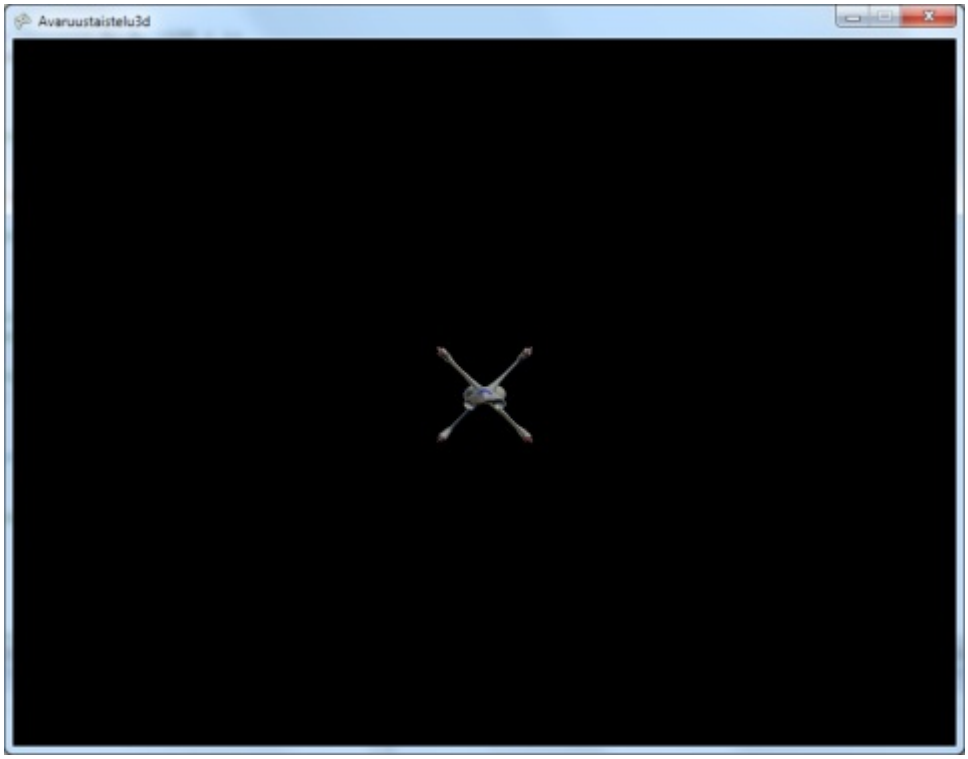
```
GameObject3d enemyShip;
```

```
protected override void LoadContent()
{
  spriteBatch = new SpriteBatch( GraphicsDevice );

  enemyShip = new GameObject3d( new Vector3( 0, 0, -100 ) );
  enemyShip.Model = Content.Load<Model>( "spaceship" );
}
```

```
protected override void Draw( GameTime gameTime )
{
  GraphicsDevice.Clear( Color.Black );
  enemyShip.Draw( camera );
  base.Draw( gameTime );
}
```

Pelin pitäisi nyt näyttää jotakuinkin tältä:



Kuva 14: 3d-alus sadan yksikön päässä kamerasta.

9.4 Kameran pyörittäminen

Jotta voisimme katsella ympärillemme 3d-maailmassa, täytyy kameraa saada myös käännellä. Jos kameraa tarvitsisi kiertää vain yhden kiinteän akselin ympäri, yksi muunnosmatriisi riittäisi, mutta kahden tai useamman akselin tapauksessa tulee mutka matkaan. Komponenttiakselien ympäri kiertävät matriisit voidaan kyllä kertoa keskenään, mutta kierrot vaikuttavat toisiinsa hankalasti ennustettavalla tavalla. Tämä *gimbal lockiksi* kutsuttu ongelma saadaan kuitenkin ratkaistua käyttämällä *kvaternioita*, joille XNA:ssa onneksemme on oma luokkansa. Kvaternio itsessään koostuu neljästä reaaliluvusta, jotka määrittelevät akselin ja sen ympäri tehtävän kierron, mutta niiden laskutoimitusten syvällisempi ymmärtäminen vaatii korkeamman tason matematiikkaa kuin aloittelevalta peliohjelmoijalta voidaan vaatia. (Grootjans 2009, s. 50-51)

Lisäämme siis kameraluokkaan kiertokvaternion ja alustamme sen `Initialize`-metodissa. Alkuarvoksi määritelty `Quaternion.Identity` eli yksikkökvaternio tarkoittaa, ettei mitään kiertoa ole tehty.

```
public Quaternion Rotation { get; private set; }
```

```
public override void Initialize()
{
  Projection = Matrix.CreatePerspectiveFieldOfView(
    MathHelper.PiOver4,
    Game.GraphicsDevice.Viewport.AspectRatio,
    1, 3000 );

  Rotation = Quaternion.Identity;
  UpdateView();

  base.Initialize();
}
```

Kvaterniota voidaan käyttää `UpdateView`-metodissa päivittämään näkymämatriisin ja vektorit `Forward` ja `Up`.

```
private void UpdateView()
{
  Vector3 origTarget = new Vector3( 0, 0, -1 );
  Vector3 origUp = new Vector3( 0, 1, 0 );
```

```
  Vector3 rotTarget = Vector3.Transform(
    origTarget, Rotation );
  Vector3 target = Position + rotTarget;

  Up = Vector3.Normalize( Vector3.Transform(
    origUp, Rotation ) );
  Forward = Vector3.Normalize( target - Position );
  View = Matrix.CreateLookAt( Position, target, Up );
}
```

Itse `Rotate`-metodista tulee näin varsin yksinkertainen. Muunnokset voidaan yhdistää yhdeksi kvaternioksi kertomalla ne keskenään, jonka jälkeen itse kierto kerrotaan yhdistetyllä muunnoksella. Toisin kuin matriisien kertolaskussa, kvaternioiden kertolaskussa muunnokset suoritetaan vasemmalta oikealle. (Grootjans 2009, s. 52)

```
public void Rotate( float yaw, float pitch, float roll )
{
  Quaternion newRotation =
    Quaternion.CreateFromAxisAngle( Vector3.UnitY, yaw ) *
    Quaternion.CreateFromAxisAngle( Vector3.UnitX, pitch ) *
    Quaternion.CreateFromAxisAngle( Vector3.UnitZ, roll );

  Rotation = Rotation * newRotation;
}
```

Nyt meillä on valmis metodi kameran pyöritykseen. Käyttäen luvun 6.6 hiiren suhteellisen liikkeen kuuntelua, voimme käännellä kameraa ympäriinsä hiirtä heiluttamalla. Kameran pyöritysnopeutta voidaan hienosäätää kertomalla hiiren paikan muutoksia vakiolla.

```
int centerX, centerY;

protected override void Initialize()
{
  centerX = Window.ClientBounds.Width / 2;
  centerY = Window.ClientBounds.Height / 2;
  Mouse.SetPosition( centerX, centerY );
  // ...
```

```
}

protected override void Update( GameTime gameTime )
{
  MouseState mstate = Mouse.GetState();
  int yaw = mstate.X - centerX;
  int pitch = mstate.Y - centerY;
  camera.Rotate( -MathHelper.PiOver4 / 150 * yaw,
                 -MathHelper.PiOver4 / 150 * pitch, 0 );
  Mouse.SetPosition( centerX, centerY );
  //...
}
```

9.5 Kameran liikuttaminen

2d-pelissä aikaisemmin käyttämäämme portaittaista nopeudensäätöä voidaan käyttää myös 3d-pelissä pienillä muutoksilla. Tässä tapauksessa liikutettava on aluksen sijaan kamera, ja kulman määräämän suunnan sijaan se liikkuu oman `Forward`-vektorinsa suuntaan tai sitä vastaan. Paikan päivittäminen pelin `Update`-metodissa on täysin toimiva tapa, mutta siistimpi ratkaisu on lisätä kameralle nopeus ominaisuutena.

```
public float Speed { get; set; }
```

```
public override void Update( GameTime gameTime )
{
  UpdatePosition( gameTime );
  UpdateView();
  base.Update( gameTime );
}

private void UpdatePosition( GameTime gameTime )
{
  double dt = gameTime.ElapsedGameTime.TotalSeconds;

  if ( dt <= 0 || Speed == 0 )
    return;
```

```
    Position += (float)(Speed * dt) * Forward;
}
```

Näin kamera liikkuu eteenpäin niin monta yksikköä sekunnissa kuin sen nopeus määrää. Nopeuden muuttaminen näppäimistöllä tai peliohjaimella onnistuu täsmälleen samaan tapaan kuin 2d-esimerkissä.

9.6 2d-grafiikkaa 3d-maailmassa - HUD (Heads Up Display)

Alun perin ilmailusta peräisin olevalla termillä *HUD* (*Heads Up Display*) tarkoitetaan ohjaamon etuikkunaan projisoitavaa ”näyttöä”, jossa näkyy tähtäysristikko ja tietoja esimerkiksi näkyvissä olevasta kohteesta. HUD on yleinen näky myös 3d-peleissä, joissa se näkymän päälle piirrettynä tähtäimineen paljastaa esimerkiksi käytössä olevan aseen ammuksien määrän, pelaajan terveyden tai muuta nopeasti muuttuvaa tietoa. Tässä alaluvussa lisäämme peliin tällaisen HUDin, johon piirrämme tähtäinristikon ampumista varten.

HUD-grafiikan piirtäminen ei eroa mitenkään aiemmin käytetystä tavasta piirtää 2d-grafiikkaa. Kun tähtäysristikon kuva on lisätty projektiin, voidaan se ladata muuttujaan `LoadContent`-metodissa ja piirtää ruudulle `Draw`-metodissa `SpriteBatch`ia käyttäen. On kuitenkin oltava tarkkana, että lisää 2d-piirtokoodin viimeiseksi, muuten 3d-näkymä peittää koko HUDin näkyvistä.

```
Texture2D crosshair;
Vector2 crosshairPos;

protected override void LoadContent()
{
  // ...
  crosshair = Content.Load<Texture2D>( "Crosshair" );
  crosshairPos = new Vector2(
    centerX - crosshair.Width / 2,
    centerY - crosshair.Height / 2 );
}

protected override void Draw( GameTime gameTime )
{
  GraphicsDevice.Clear( Color.Black );
  enemyShip.Draw( camera );
```

```
    spriteBatch.Begin();
    spriteBatch.Draw( crosshair, crosshairPos, Color.White );
    spriteBatch.End();

    base.Draw( gameTime );
}
```

10 Yhteenveto

XNA on erinomainen työkalu aloittelevalle peliohjelmoijalle. Content-järjestelmän suoraviivaisuus ja helpommin lähestyttävä 3d-ohjelmointi ovat sen suurimpia valtteja. Yksinkertaistamisen varaa olisi kuitenkin edelleen, esimerkkinä näppäimistön ja peliohjainten kuuntelun muuttaminen tapahtumapohjaiseksi pollauksen sijaan. Valmiille ajastinluokalle ja animoidulle tekstuurille olisi myös käyttöä.

Osaavammalle ohjelmoijalle XNA tarjoaa lähinnä mahdollisuuden tehdä nopeita prototyyppejä ja Xbox 360 -pelejä, jos on valmis maksamaan Creators Club -jäsenyydestä. Toisaalta XNA:sta löytyy käytännössä samat ominaisuudet kuin alla olevasta DirectX:stäkin matalimman tason laitteisto-ominaisuuksia lukuunottamatta. Jos on valmis maksamaan ohjelmointimukavuudesta pienellä pudotuksella vasteaikoihin ja hieman rajoittavammalla lisenssillä, voi XNA olla perusteltu valinta kokeneemmallekin koodarille.

Pelin tekeminen Xbox 360 -alustalle on tehty helpoksi, mutta sadan dollarin maksaminen vuodessa oikeudesta pelata omatekoisia pelejä yhdellä konsolilla voi helposti tuntua kohtuuttomalta. Sen sijaan jos aikoo laittaa hengentuotteensa myyntiin eikä ole onnistunut vielä saamaan julkaisijoiden huomiota, oikeus Xbox Live Marketplacen käyttöön voi tehdä siitä hyvänkin sijoituksen.

Zunea ei ole Suomessa myynnissä, mutta versioon 4 luvattu Windows 7 Phone -älypuhelintuki voi nostaa XNA:n arvoa useamman alustan peliohjelmointiympäristönä.

Tämän tutkielman kirjoitushetkellä merkittäviä kaupallisia XNA:lla toteutettuja pelejä ei ole tullut vastaan, poislukien Xbox Live Marketplacen pelit. Niistäkin suurin osa on retrohenkisiä 2d-pelejä. Suuria kaupallisia XNA:lla tehtyjä Xbox 360 -pelejä tuskin tullaan näkemään, ellei Microsoft löyhennä käyttöoikeussopimustaan. PC:llä tähän on kuitenkin kaikki mahdollisuudet, jos vain kaupallisten pelien ja fysiikkamoottorien kehittäjät näkevät XNA:n sopivaksi tarkoituksiinsa.

Lähteet

Carter, C. 2009. *Microsoft XNA Game Studio 3.0 Unleashed*. Indianapolis: Sams.

Grootjans, R. 2009. *XNA 3.0 Game Programming Recipes — A Problem-Solution Approach*. Berkeley: Apress.

Hejlsberg, A., Wiltamuth, S. & Golde, P. 2006. *The C# Programming Language*. Crawfordsville: Pearson Education.

Jyväskylän yliopisto, 2010. *Jypeli-peliohjelmointikirjasto*. Saatavilla WWW-muodossa <URL: `https://trac.cc.jyu.fi/projects/npo`>. Viitattu 7.10.2010.

Microsoft, 2010. *Microsoft Software License Terms, Microsoft XNA Game Studio 3.1*. Saatavilla asennusohjelman yhteydessä: <URL: `http://www.microsoft.com/downloads/en/details.aspx?FamilyID=53867a2a-e249-4560-8011-98eb3e799ef2`. Viitattu 7.10.2010.

Microsoft, 2010. *XNA Game Studio 3.1, Programming Guide*. Saatavilla WWW-muodossa <URL: `http://msdn.microsoft.com/en-us/library/bb198548.aspx`>. Viitattu 7.10.2010.

Miles, R. 2009. *Introduction to Programming through Game Development Using Microsoft XNA Game Studio*. Saatavilla WWW-muodossa <URL: `http://www.microsoft.com/education/facultyconnection/articles/articledetails.aspx?cid=1987`>. Viitattu 7.10.2010.

Reed, A. 2009. *Learning XNA 3.0*. Sebastopol: O'Reilly.