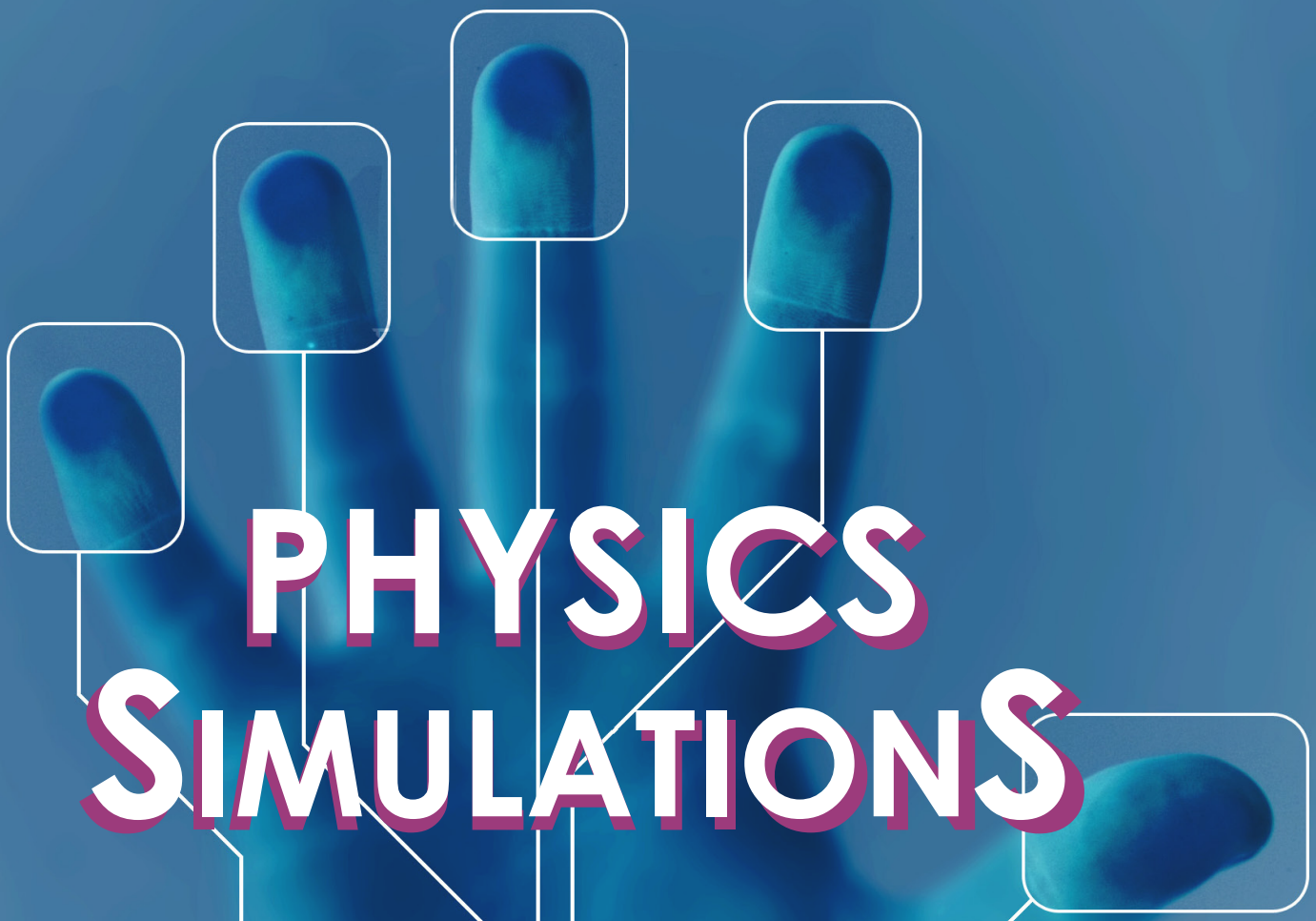
Game Coder Magazine

PHYSICS



PHYSICS SIMULATIONS

Fluid Simulation Using SPH

Introduction to 3D Collision Detection

Creating a Soft Body System

Notes on Simple Rigid Body Dynamics

Issue 01/2012 (1)

PLUS

>>> UNITY 3D >>> Definition by Aggregeration:
An Introduction to the Entity Component-Model for Games Development

Dear Readers,

Many of you mentioned, that creating a magazine about Game Physics is a brilliant idea. Today, we are presenting you the very first issue of Game Coder Mag Physics! To be more specific, the magazine is devoted to programming the physics engine: including object movement, collision detection, physics simulation etc. If you'd like to see a specific topic in our magazine, don't hesitate and drop us a line at en@gamecodermag.com. We'll take your proposition under consideration, and you can help us to develop this new creation for people passionate in game development!

Now, let's look through the content:

This issue is devoted to creating basic physics simulations such as: rigid body simulation, fluid simulation using SPH, soft body dynamics and much more.

Soft body dynamics focuses on visually realistic physical simulations of the motion and properties of deformable objects. If you want to make your creation more realistic, don't forget about adding clothes or hair.

Creating a Soft Body System by Soufiane Khiat will be very helpful. You'll learn how to improve your physics engine with an implementation of soft body system. Check it out!

As an extension of making things moved like in reality, I highly recommended an article by Reza Ghvami. If you want to learn an ideal method for simulating a sufficiently accurate fluid that performs at an acceptable frame-rate you can't miss Fluid Simulation using SPH.

We cannot also forget about one of the most basic simulation using in games: Rigid Body. Todd Seiler's tips will help you to develop a simple Rigid Body Dynamics simulation. From engine structure's components through Quaternions, numerical precision, frame, time steps and much more, to using an Integrator as way to update the state of every game object - have fun!

For begginers I also recomended Introduction to 3D Collision Detection by Ruben Alcaniz. This short introduction will bring you into the world of colliders, coarse collision and collision detection algorithms.

You can also get a brief outline of how to develop an ECM game engine in C# using the XNA framework. Undoubtedly, Definition by Aggregeration: An Introduction to the Entity Component-Model for Games Development by Christopher Mann will give you some useful information on this topic.

In Plus Section we have a short summary on Unity 3D Interface. The article shows the power of Unity 3D - one of the unique tools, used for creating 3D applications, films and other content such as: architectural visualizations, real time 3D animations or simulations.

I hope you enjoy the very first issue of Game Coder Mag Physics!

Agnieszka Szałaj & Game Coder Mag Team

Managing: Agnieszka Szałaj
agnieszka.szalaj@software.com.pl

Senior Consultant/Publisher: Paweł Marciniak

Editor in Chief: Grzegorz Tabaka
grzegorz.tabaka@software.com.pl

Art Director: Anna Wojtarowicz
anna.wojtarowicz@software.com.pl

DTP: Anna Wojtarowicz
anna.wojtarowicz@software.com.pl

Production Director: Andrzej Kuca
andrzej.kuca@software.com.pl

Marketing Director: Grzegorz Tabaka
grzegorz.tabaka@software.com.pl

Beta testers: Gregory Chrysanthou, Humberto Sanchez, Dawid Esterhuizen, Brett Ragozzine

Publisher: Software Media Sp. z o.o. SK
02-682 Warszawa, ul. Bokserska 1
www.gamecodermag.com

Creating a Soft Body System

by Soufiane KHIAT

In modern AAA game engine, reality is the best model. In this article we'll show how to improve your physic engine with an implementation of soft body system. The soft body can be clothes, hair, jelly, or any deformable object. This deformation can create a realism feeling.

Introduction

To create our soft body system we need to lay the basis. The soft bodies are an aggregation of particles and springs. The springs are used to link the particles between themselves. This little trick allows us to create a realistic soft body. Cloth is a plane of mass-spring; ropes are lines of mass-springs… In theory it should be very easy, but the implementation can be a real nightmare.

Mathematics & Physics

The springs and particles are the basic elements of the soft bodies. Indeed our system is subject to gravity and many constraints. For example if I drop a cloth I would like this cloth to be influenced by gravity, wind and other forces. As we cannot predict all the interactions in a game, we cannot have an explicit function for springs and particles behaviors. So we will use Verlet[1] integration system, and in this article we will not discuss about any other numerical solver[2].

The formula of the Verlet method is:

$$x_{t+\Delta t} = 2x_t + x_{t-\Delta t} + a_t \Delta t^2 + O(\Delta t^4)$$

As we can see, the formula is very explicit. We note 'x' as the position of the particle, 'a' as the acceleration, 't' the current time and Δt as the differential time (the time between 2 frames of computing). So the next position depends only on the current and the old position and the current acceleration of the particle.

So in the C++' source code we have a little structure to model our particle:

```
struct Particle
{
f32 fMass;
Vector3 vPosition;
Vector3 vOldPosition;
Vector3 vAcceleration;
bool bMoveable;
};
```

This particle will be stored in a list, but you can save your particles in a random access data structure.

The behavior of springs is governed by the Hook law:

$$F = k_s \Delta l$$

As we can see the force only depends on a constant ks (this constant describes the spring stiffness) and variation of length (Δl). So (**in order**) to model our spring we only need a reference of 2 particles and the rest's length.

```
struct Spring
{
f32 fLength;
f32 fKs;
f32 fKd;
Particle* pParticleA;
Particle* pParticleB;
};
```

If you store your springs in a random access data structure you can use an **unsigned int** to save the index in the array, to know where the particle is.

We will suggest an interface (code) to manipulate this element (Listing 1).

The solving of the system can be sequenced in 4 distinct steps(Figure 1):

1. Apply Gravity: just set forces: a=-9.81j
2. Compute Spring : Apply Hook and damping forces: F=ksΔl -kddxdt
3. Collision: Solve collision with ground & spheres
4. Verlet Integrate: Integration forces to have position of particles

The first difficulty in the implementation of any physical system not the implementation itself. The problem is the parameterization of the system. How to choose the value of ks or Δt[3] ?

To find an answer to this problematic we will show the unit of the equation (In this article we will use the International System - IS).

With this equation: F=ma ; we have F that in Newton (N⇔kg/(m.s-2)), m in kg (kilogram) and a in m.s-2.

So with this analysis, we can find the unit of ks. So ks is in N.m-1. To parameterize our system we have to find a range of value for the

1 Loup Verlet http://en.wikipedia.org/wiki/Loup_Verlet

2 if you are interested you can see Euler, Runge Kutta 2 and 4…

3 The rendering and physics calculus could be in asynchrony so the Δt could be different of the Δt of renderer

stiffness constant. As this parameter is in N.m-1 we understand that it is a force by distance. Our springs must withstand the gravity, so ks needs to be at least greater than gravity force (9.81). But that is in theory, in practice if you use a big value like 9 it produces a huge instability of our system. So in the real world a parameterization of physics engine is an art and not a real science. For example the value used for cloth (2d plane of soft body) ks=0.00625 N.m-1 & kd=10-6 kg.m.s-2. This value is not usable for the 3d soft body, we can use ks=0.00125 N.m-1 & kd=10-7 kg.m.s-2.

A problem for the new C++ programmers is the no consideration of the implicit constructions and conversions of the objects. This could be a big problem for the performances (in terms of performance). To solve that we try to reduce the implicit conversion and we use many inline functions to reduce the number of implicit creations of temporary variables.

Technical pause

In this short code:

```
Vector3 vVector2 = 2.f*vVector3 +
( 3.f*vVector5 ).CrossProduct
( vVector7 );
```

In this simple code we have 2 implicit constructions:

- 2.f*vVector3 : Construction
- 3.f*vVector5 : Construction
- CrossProduct : Construction
- + : Construction

With more code we can reduce the construction:

```
Vector3
vVector2 = vVector5;
vVector2 *= 3.f;
vVector2 ^= vVector7; // Cross Equal
vVector2 += 2.f*vVector3;
```

More code, the same result but only 1 construction:

```
2.f*vVector3 : Construction
```

So I say that to justify the next code.

Listing 1

```
class MassSpringModel
{
public:
MassSpringModel( f32 fDTime = 0.0125f );
~MassSpringModel();
Sphere* AddCollider(  f32 const fMass, f32 const fX, f32 const
fY, f32 const fZ, f32 const fRadius );
Particle* AddParticle( f32 const fMass, f32 const fX, f32 const
fY, f32 const fZ, bool bMoveable = true );
Spring* AddSpring( f32 const fKs, f32 const fKd, f32 const
fLength, Particle* const pParticleA, Particle* const pParticleB
);
void GenerateAllSpring( f32 const fRestLength, f32 const fKs,
f32 const fKd );
ParticlesPair GenerateRope( u32 const uCountElement, f32 const
fLengthElement, f32 const fKs, f32 const fKd, f32 const fMass
);
ParticlesPair GenerateCloth( u32 const uDepth, u32 uX, u32 uY,
f32 const fDx, f32 const fDy, f32 const fMass, f32 const fKs,
f32 const fKd );
Particle* GenerateGrid( u32 const uDepth, u32 uX, u32 uY, u32
uZ, f32 const fDx, f32 const fDy, f32 const fDz, f32 const
fMass, f32 const fKs, f32 const fKd );
void Compute();
ParticlesList* GetParticlesList();
SpringsList* GetSpringsList();
SpheresList* GetSpheresList();
void ComputeSpring();
void ComputeGravity();
void ComputeCollision();
void Integrate();
private:
f32 m_fDTime;
ParticlesList m_lParticlesList;
SpringsList m_lSpringsList;
SpheresList m_lSpheresColliderList;};
```

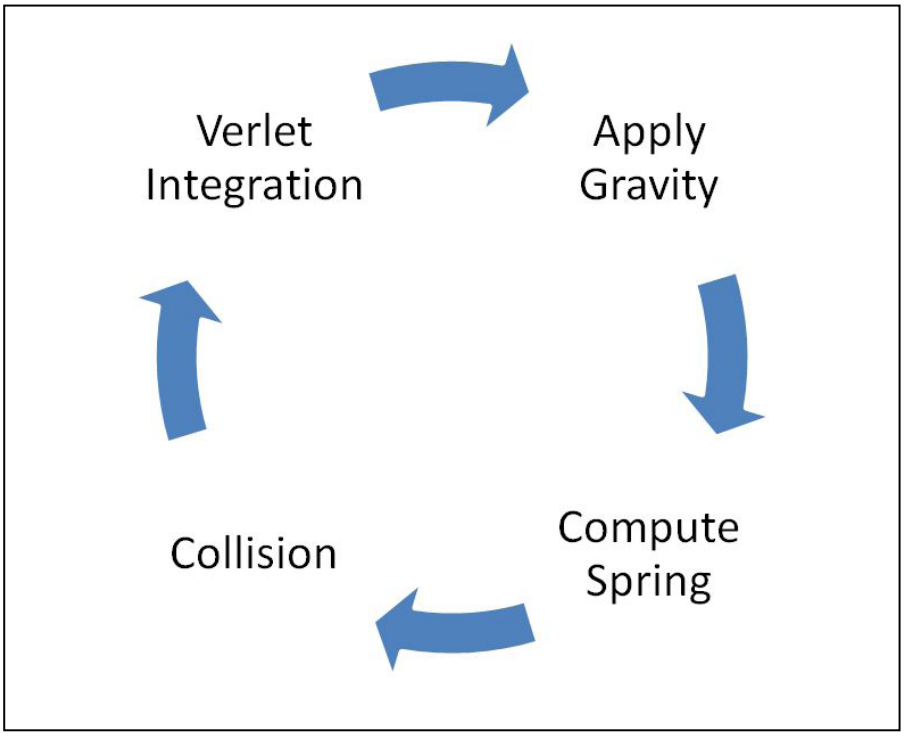


Figure1: *The solving of the system in 4 distinct steps*

Listing 2

```
vDelta = pSpring->pParticleB->vPosition;
vDelta -= pSpring->pParticleA->vPosition;
f32 fDelta = vDelta.Length();
vDelta.Normalize();
f32 fDiff = ( fDelta - pSpring->fLength )/
fDelta;
vOffset = vDelta;
vOffset *= 0.5f*fDiff*pSpring->fKs;
if ( pSpring->pParticleA->bMoveable )
pSpring->pParticleA->vPosition += vOffset;
if ( pSpring->pParticleB->bMoveable )
pSpring->pParticleB->vPosition -= vOffset;
```

Listing 3

```
// Damping force
vSpeedA = pSpring->pParticleA->vPosition;
vSpeedA -= pSpring->pParticleA->vOldPosition;
vSpeedA *= 0.5f*pSpring->fKd/m_fDTime;
vSpeedB = pSpring->pParticleB->vPosition;
vSpeedB -= pSpring->pParticleB->vOldPosition;
vSpeedB *= 0.5f*pSpring->fKd/m_fDTime;

if ( pSpring->pParticleA->bMoveable )
pSpring->pParticleA->vPosition += vSpeedA;
if ( pSpring->pParticleB->bMoveable )
pSpring->pParticleB->vPosition -= vSpeedB;
```

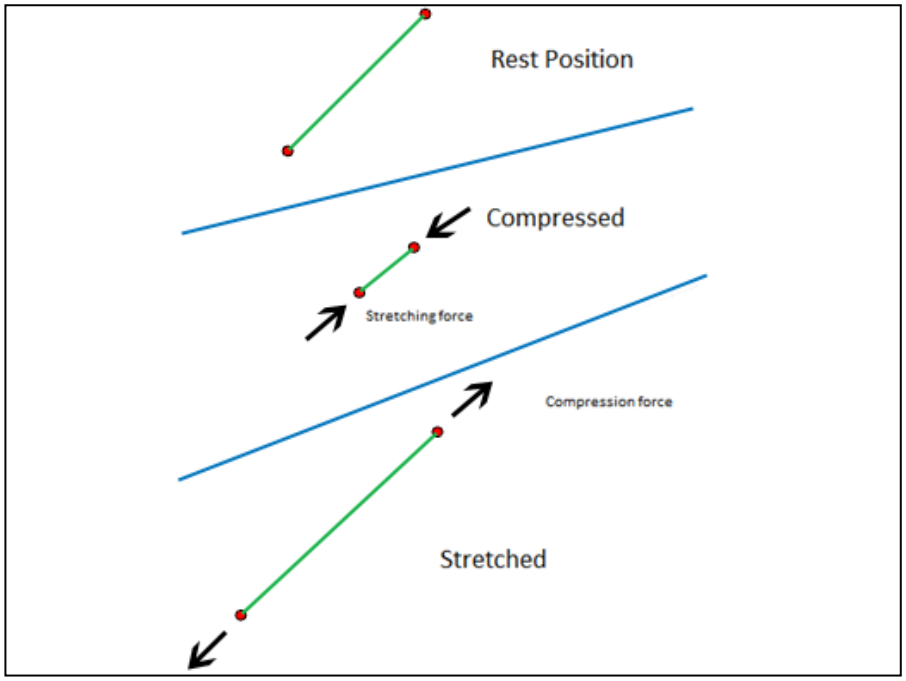


Figure 2

Principia

To understand it, we have to understand, the principle behind the springs before: (Figure 2).

If a spring is compressed, the Hook's force tries to stretch it and if the spring is stretched the Hook's force tries to compress the spring. A force has an axis and a direction. The axis is along the spring (the segment between particles) and the direction depends on compression or stretch state. The direction of a particle is the opposite of the other particle in the same spring.

To compute spring, this code is for each spring: (Listing 2).

To find the direction we use the vDelta vector. The sign of fDiff helps us to adjust the direction and magnitude of Hook's force. As you can see, vOffset is divided by 2 to be applied in the 2 particles.

As you notice it we do not use a force to apply it on a particle. We do that because if you use the force and solve the physic equations we can have a big numerical instability! So to keep ourselves from that, we apply directly the forces as motion; of course it isn't correct in physics. But in video games the reality is just an inspiration not a goal.

If we try this solution, many soft bodies can oscillate forever. To minimize this effect we have to add the damping force, the equation can be written as follow:

$$Fd=-kddxdt$$

The member dxdt represents the speed so the unit is in m.s-1. The real unit of the force is kg.m.s-2; so the unit of kd constant is kg.s-1. To compute dxdt we can use the old position as follow:

$$Fd=-kdPosition-PositionoldΔt$$

In C++ for each spring by particle: (Listing 3).

We use the same trick: do not use the force application but displace the particle. So the integration is just for gravity application and implicit reaction of the spring.

The value of kd depends on ks and on masses of particles. Now to have our simulation we need body creating. To begin with a practical example we will implement the simplest soft body as possible, a rope! To create a rope it is very easy with our system. As you can see it in the following code: (Listing 4).

Note the trick to create a fixed joint, indeed we removed the first particle created. If a particle is not in the m_lParticlesList this one will not receive a gravity force. As this particle is not included in the list, when we solve our system during the integration the position of this particle doesn't updated.

To interact with the environment as an example we add a collision with sphere. It is not a real collision with the soft body but only with the particle, it's enough for our utilization use[4].

4 you can easily add collision with AABB, OBB…

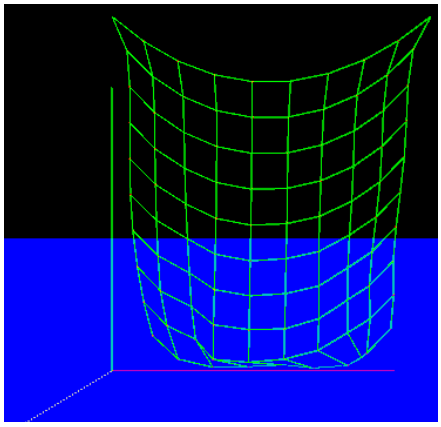


Figure 3: *A simple grid of spring*

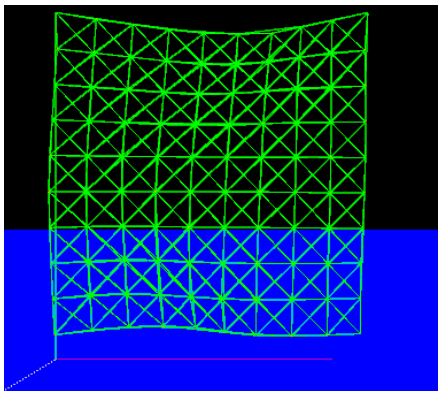


Figure 4: *Structure of mass-spring system*

To simplify the simulation and code a sphere collider is only a particle with a radius:

```
struct Sphere
{
Particle* pParticle;
f32 fRadius;
};
```

With this solution we can add a sphere and use it as a particle and motion with external forces (gravity, wind, spring…) could be used directly without adding any code. If I want a sphere collider attached on a spring or a fixed joint because it is already implemented. To have our "complete" collision system we just need to add a collision:

- Particle/Ground
- Sphere/Ground
- Particle/Sphere
- Sphere/Sphere

This article is focused in (**on**) soft body implementation so if you want to implement it in real case you have to use a technique to reduce test number: BSP, Octree…

Now we can increase a little bit the complexity with the implementation of clothes. The naïve solution is a simple grid of spring. But with this structure we will have this behavior (Figure 3):

It is not a realistic behavior of cloth, to fix this behavior we will add a shear spring. And with this structure of our mass-spring system we will have a more realistic simulation as you can see it: (Figure 4).

To be more precise we structure our system as follow: (Figure 5).

Go Further

That is the basis of soft body simulation. To go further you can add wind force, breakable springs... It is really complicated. The wind force depends on the surface so we need a triangle element, the breakable spring(**s**) depends on current forces in spring of elongation… It is very easy to improve this system.

The core difficulty is the parallelization, collision optimization and self-collision of soft bodies. The cards are in your hands.

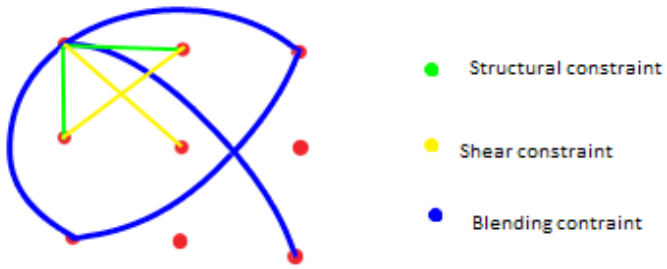


Figure 5

Listing 4

```
ParticlesPair MassSpringModel::GenerateRope( u32 const uCountElement, f32 const
fLengthElement, f32 const fKs, f32 const fKd, f32 const fMass )
{
ParticlesPair oRet;
f32 const fRopeLength = static_cast< f32 >( uCountElement )*fLengthElement + 1.f;
f32 fCurrentPosition = 0.f;
Particle* pOldParticle = AddParticle( 1.f, fCurrentPosition, fRopeLength, 0.f, false );
oRet.first = pOldParticle;
Particle* pCurrentParticle = 0x0;
for ( u32 uElement = 0; uElement < uCountElement; ++uElement )
{
fCurrentPosition += fLengthElement;
pCurrentParticle = AddParticle( fMass, fCurrentPosition, fRopeLength, 0.f );
AddSpring( fKs, fKd, fLengthElement, pOldParticle, pCurrentParticle );
pOldParticle = pCurrentParticle;
}
oRet.second = pCurrentParticle;
m_lParticlesList.remove( oRet.first );
return oRet;  }
```

Fluid Simulation Using SPH

by Reza Ghavami

Parallel computing on multiple core GPUs has unlocked exciting solutions, as well as some new challenges, in the simulation of fluids in games. Impressive processing power, however, is not enough to create an interactive, real-time simulation of fluids at a convincing frame-rate for the end user. Fortunately, in today's games, optimum performance is more important than simulation accuracy. Modeling the fluid as a system of particles proves to be an ideal method for simulating a sufficiently accurate fluid that performs at an acceptable frame-rate. This method is called smoothed particle hydrodynamics (SPH). Figure 1 illustrates an example of what a particle-based fluid looks like in the game, *Batman: Arkham Asylum*.

SPH Equations

In a system of n fluid particles where $\{i : 1 \leq i \leq n\}$, each particle has position, $\mathbf{x_i}$, velocity, $\mathbf{v_i}$, acceleration $\mathbf{a_i}$, mass mi, and density ρ_i. When using the general SPH equation, these quantities can be described by the general scalar quantity A. By definition

$$f^p(x) = -\nabla p(x) = -\sum_{j=1}^{n} \frac{m_j}{\rho_j} p(x) \nabla W(x - x_j, h)$$

(I)

where *W* is a scalar-valued function called a smoothing kernel [Müller 03]. Quantities for *A* can be found by interpolation at any location xi. Mass *mj* is constant for all particles, while the density ρ*j can vary for each particle during the simulation.*

Using the Navier-Stokes equations, in addition to conservation of mass and momentum, the force on each particle due to pressure [Müller 03] is derived as

$$f_i^p(x) = -\nabla p_i(x) = -\frac{m_i}{\rho_i} \sum_{j=1}^{n} \frac{m_j}{\rho_j} \frac{[(p]_i + p_j)}{2} \nabla W(x - x_j, h)$$

(II)

Taking into account the symmetry of pressure forces between any pair of particles *i and j, the value* (pi + pj)/2 is substituted for p(x). With the acceleration of the fluid defined as a(x) = f(x)/ρ(x), the per-particle force fpi becomes

$$\text{"f"}_i\text{"i"}^{\uparrow}\text{"p"}\ (\text{"x"}) = \text{"}- \ [\nabla p]_i\ (\text{"x"})\text{"} = -m_i/\rho_i \sum_{j=1}^{n} m_j/\rho_j \ (p]_i + p_j))/2 \ \nabla W(\text{"x"} - \text{"x"}_j\text{"j"}, h)$$

(III)

Similar to the equation for the pressure force, $\mathbf{f}^p{}_i$, the viscous force is approximated as

$$f_i^v(x) = \mu \nabla^2 v = \mu \frac{m_i}{\rho_i} \sum_{j=1}^{n} \frac{m_j}{\rho_j} (v_j - v_i) \nabla^2 W(x - x_j, h)$$

(IV)

where μ is the viscosity constant.

The Importance of Rest Density

In a particle-based simulation of incompressible fluids such as water or oil, it is preferable to model the fluid as a compressible one [Green 11]. The general SPH equation, Equation (i), drives the particles towards a high-density state. While in this high-density state, the fluid gives the illusion that it is incompressible, as in Figure 2. After collision, the objective is to move the particles to a state called rest density.

Particles in a fluid should collide with objects such as walls, terrain and characters, in a way that makes the interaction look realistic to the user. This is evident in Figure 1 as the steam interacts with the Batman character. The density at each particle depends on the number of neighboring particles within the

Figure 1: *Batman walking through steam in Batman: Arkham Asylum [Copyright NVIDIA Corporation, Eidos Interactive and Rocksteady Studios]*

radius *h* from the SPH equations. In other words, the closer the particle is to its neighbors, the higher the recorded density for that particle. Figure 3 shows that the density field at each particle can be visualised graphically as a Guassian shaped splat at each particle position.

Using the equation of state

$$p = k(\rho - \rho_0),$$

the pressure at each particle can then be calculated. Forces are then applied to move the particles from areas of high pressure to areas of low pressure. A fluid colliding with an object has the tendency to bunch up near the object, so forces must then be applied to move the fluid away from the object to restore the rest density. This creates the splash effect, which makes the scene more realistic. Figure 4 depicts the results of this action graphically.

Data Structures: Spatial Hashing

Fluid simulation can require the processing of thousands of particles, which makes the choice of data structure for handling the particles crucial. For optimum simulation performance, the spatial hash data structure is found to be preferable. A spatial hash consists of a hash table, represented by the horizontal row of boxes in Figure 5. Each entry of the hash table is known as a hash bucket. The hash buckets, shown as vertical columns of boxes, are essentially lists of fluid particles. These particles are indexed to the hash table according to their positions $\mathbf{x}_i$ and the hash function $H(\mathbf{x})$.

Such a spatial hash requires *O(n)* computation to construct, while a query of the structure is of the order *O(q)*, where *q* is the number of fluid particles mapped to a single hash bucket [van Kooten 10].

Two main phases are involved in the performance of a spatial hash when processing fluid particles: the construction phase and the query phase. Immediately after the fluid

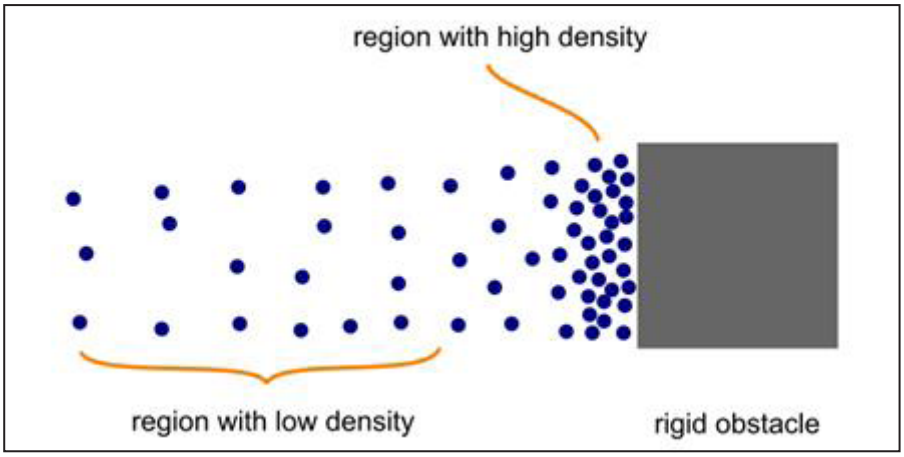


Figure 2: *Fluid particles colliding with a game object*

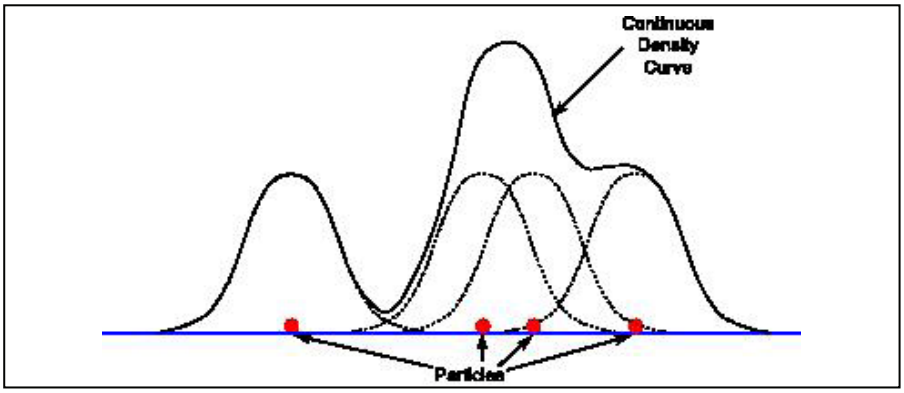


Figure 3: *Guassian splat that shows density field of a particle-based fluid. The x axis is distance from the origin, and y axis is the density. Notice the density is highest when the particles are closest together*

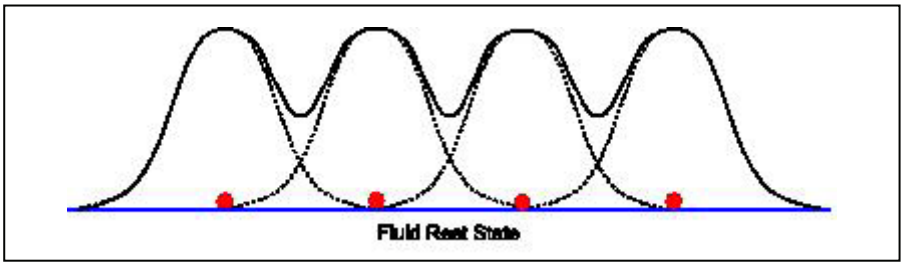


Figure 4: A fluid that has returned to rest density, after colliding with an object. As in the previous graph, the x axis is distance from the origin, and y axis is the density.

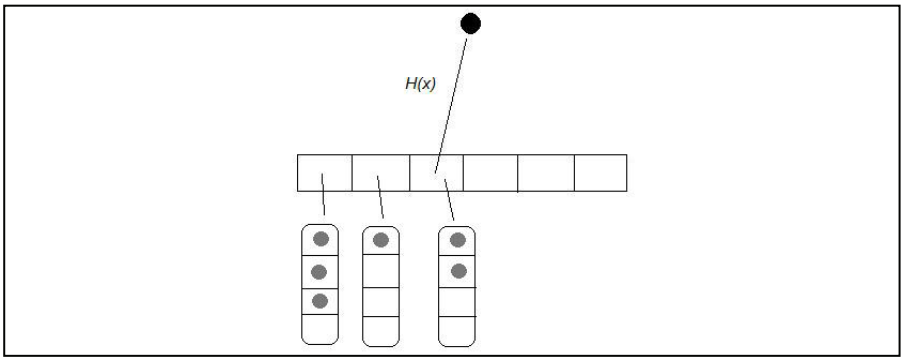


Figure 5: *The structure of a spatial hash. The black dot at the top represents a fluid particle,mapped to the hash table below it by the function H(x). Each hash bucket, in turn, points to a list of fluid particles.*

particle positions are found, the hash structure can then be constructed in the construction phase. This is followed by the query phase, in which the density and force calculations are made. To efficiently construct the spatial hash on a parallel architecture, the particles belonging to the same hash bucket should be kept in a contiguous block of memory, to avoid cache misses during the query phase.

A spatial hash is constructed either from scratch each frame or incrementally. Construction from scratch requires sorting the particles based on hash index. A simple iteration through the lists of particles is done to find the start of each hash bucket. Constructing a spatial hash incrementally involves preallocating the hash buckets based on a maximum number of particles per hash bucket. Care must be taken, however, not to exceed the maximum number of particles estimated during this preallocation.

The SPH Algorithm

Solving the SPH equations results in the particle positions ($[\mathbf{x}_i]^0$, $[\mathbf{x}_i]^1$, $[\mathbf{x}_i]^2$, …) at the time steps (t_0, t_1, t_2, …). Here the bracket notation for any quantity $[A]^n$ represents the value of A at time t_n. With Equations (i) – (iv), the densities, forces and accelerations of the particles can be calculated in specific steps described by Algorithm 1 below. This algorithm sacrifices accuracy for the sake of computational efficiency. The algorithm, written in pseudocode, is the following: (Algorithm 1).

Performance Data

The performance of the SPH algorithm is put to the test by using a water-in-a-glass simulation, implemented on a single core of an Intel Xeon W3520 processor. The simulation consists of three main steps: in step 1, the water is poured and splashes around the glass. In step 2, the water stabilizes and comes to rest. In step 3, a wavelike formation is generated, which tends to compress the fluid particles against the sides of the glass. Figure 6 illustrates the simulation graphically. In addition, Table 1 contains measurements describing construction time of the spatial hash and evaluation of the SPH equations.

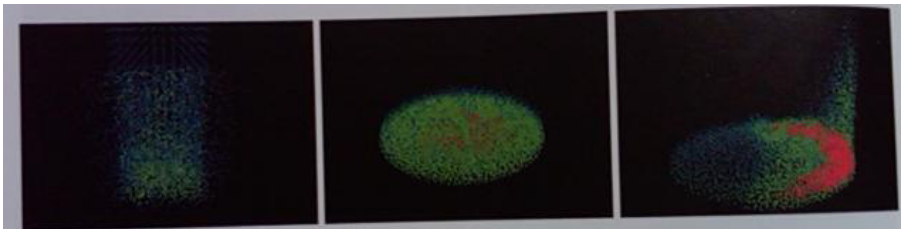


Figure 6: *The three steps measuring the performance of the SPH simulation. In Step 1, the fluid particles fall into the glass. In Step 2, the particles are at rest. In Step 3, the particles create an artificial wave. The colors indicate the density of the particles, ranging from blue (low density) to red (high density).*

Algorithm 1

```
{{      {[x_i]^0 : 1 <= i <= n}, {[v_i]^0 : 1 <= i <= n} and {[ρ_i]^-1 : 1 <= i <= n} are known
}}
begin
        construct spatial hash structure using {[x_i]^0 : 1 <= i <= n};
        for all particles i
        begin
                query spatial hash at [x_i]^0;
                for all nearby particles j from query
                begin
                        accumulate [ρ_i]^0 using [x_i]^0 and m_j;
                        accumulate [f_i]^0 using [x_i]^0, [v_j]^0, [ρ_j]^-1 and m_j;
                end;
                calculate [a_i]^0 using [f_i]^0 and m_i;
                integrate [v_i]^1 using [a_i]^0;
                integrate [x_i]^1 using [v_i]^1;
        end;
end;
```

	Scenario A		Scenario B		Scenario C	
	cons	upd	cons	upd	cons	upd
Step 1	0.806	33.2	0.938	25.5	0.974	19.7
Step 2	0.541	46.8	0.696	31.6	0.857	22.6
Step 3	0.619	54.0	0.741	33.2	0.801	25.7

Table 1: *Execution times of the SPH simulation, measured in milliseconds. The abbreviation 'cons' and 'upd' correspond to the spatial hash construction time and the simulation update time, respectively.For Scenario A, kr = 250, gr = 100, μr = 1.5; Scenario B, kr = 750, gr = 100, μr = 1.5; Scenario C, kr = 250, gr = 33, μr = 1.5; gr here is the relative gravity.*

From Table 1, it is evident that construction of the spatial hash takes much less time than evaluation of the SPH equations. Overall, the SPH algorithm yielded a framerate of 30 fps for this simulation.

Bibliography

[Green 10] Simon Green, Richard Tonge, Miguel Sainz, Dane Johnson, and David Schoemehl. "Fluid Simulation in Alice: Madness Returns." http://developer.nvidia.com/content/fluid-simulation-alice-madness-returns. 2010.

[Müller 03] Matthias Müller, David Charypar, and Markus Gross. "Particle-Based Fluid Simulation for Interactive Applications." In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 1-3. Aire-la-Ville, Switzerland: Eurographics Assocation, 2003.*

[van Kooten 10] Kees van Kooten. "Optimized SPH." In Game Physics Pearls, edited by Gino van den Bergen and Dirk Gregorius, pp.127-151. Natick, MA: A K Peters, Ltd. 2010.

strategery games

your source for everything gaming

- Indie Game Coverage
- Game Code Giveaway
- RPGs
- Wargaming
- Table Battles
- Quizzes
- Tutorials
- Daily News
- RSS Feeds
- Forum for Everything Gaming

strategerygames.com

Notes on Simple Rigid Body Dynamics

by Todd Seiler

I've compiled a small list of things you're going to need to be familiar with in order to develop a simple simulation. I intentionally left out collision detection and resolution due to the scope of this article. The first bit of information I'm going to pass on is this: get the simulation working first and then optimize the simulation later. Each simulation is different and will require a different set of data structures and algorithms. In this article, I'm focusing on the beginner. So you may, of course, want to do something different.

Types of Physics Simulations

There are several types of simulations and sometimes games will contain combinations of them. For example, Batman: Arkham Asylum has both rigid body dynamics with soft body dynamics. Batman and the objects in the world are rigid bodies, whereas Batman's cape is a cloth that flaps in the wind; soft body. Here are the most common types used in games:

1. Rigid Body: Usually consists of rigid objects, such as boxes or characters in a game.
2. Soft Body: Cloth related, Jell-O and squishy objects. They are usually represented by using a lattice of springs attached to the object in some way.
3. Fluid Dynamics: Anything related to fluids, including smoke and air since they're technically defined to be fluids. They are computationally expensive compared to rigid body simulations.

We usually don't care about other types, such as thermodynamics, relativity, quantum mechanics, etc. In this article, I'll mainly be talking about rigid body simulations.

Engine Structure - Components

One easy way to organize a physics simulation (and, partly, your engine), is to make it component based. Components are an easy way to attach and detach functionality per game object (entity). For example, an enemy game object may have a physics component attached to it (pointer to a physics component of type PhysicsComp), and this component can be detached or attached at run-time. It may also have a Collision component attached to the game object as well for for the collision detection system. Perhaps it also has a rendering component, etc. Components are flexible because they keep all the physics functionality and the state of that particular game object together in one place (inside the component).

In a game, we might have a *physics system*, that iterates though each game object, retrieves its physics component (if it has one), and calls its *Update()* function. I will define the word "update" later in the article. The Unity3D engine uses a similar component-based approach.

Quaternions

If you know what a quaternion represents, it's not scary. If you need to understand the math, well it can get involved. Technically it's defined to be a point on the surface of a unit hyper-sphere. But most people think of it as simply the "orientation." We used to use Euler angles and rotation matrices, but they have shortcomings. Quaternions are useful for a lot of reasons, such as avoiding gimbal lock, interpolation between two orientations without shearing the object, and it consumes less memory.

Essentially, a quaternion is an axis and an angle, but quaternions are interesting because they also have their own algebra. So you can multiply them together, just like matrices, and it actually means something useful (concatenating rotations together). They need to stay normalized though (having unit length) in order to represent an orientation. It can also be converted to and from a matrix, so you can just think of it just like a rotation matrix. It has four components: x,y,z,w. The x,y,z components represent the axis and the length of that axis represents the rotation around that axis. The w component is cos(angle/2) and this component is used to determine the angle. For example, if the angle is 20 degrees or 340 degrees, you can't tell the difference between the two.

Numerical precision

Don't check a float against the number 0.0f. Use a function similar to this:

```
#define EPSILON some_small_number
bool IsZero( float val )
{
    return fabs( val ) < EPSILON;
}
```

The reason has to do with numerical imprecision in floats when doing an operation. Choosing an

EPSILON usually depends on the size of the data type. What's a good epsilon then? Something small, but also something that stabilizes the simulation. You probably want to use different epsilons depending on what you're working with. For example, you probably want to use two different epsilons for checking if a vector is a zero vector and also for checking whether your objects should be put to sleep or not. Speaking of sleeping...

Sleeping Objects and Stacks

Putting objects to sleep is a common thing to do in a physics simulation because it reduces computation and resolves some other artifacts in the simulation. It works like this: after some amount of time has passed, if an object isn't moving very much, it will fall asleep. When an object is sleeping, its position and orientation are not updated. If objects never sleep, and you use a method, such as impulses, to resolve collisions, they'll slowly and continuously move around and appear to vibrate and possibly "pop" every once in a while. It doesn't look very realistic.

The object is woken up again if another moving object collides with it. Once it settles down, it's sleepy time again. If a lot of objects are all touching each other and they're sleeping, waking up one object will usually cause a chain reaction to occur and all the objects touching each other will also wake up. Unfortunately, this can cause a spike in CPU usage all at once. If there are lots of stacked objects, this is usually a cause for concern because collisions are constantly happening and objects may have great difficulty falling asleep again. Handling a lot of stacked objects efficiently is a common problem due to the large number of collisions that occur in the short period of time.

Frame

This term gets wildly abused so it's often confusing. It's used in both graphics and physics to mean two different things. To understand what this is exactly, lets imagine a game where nothing is moving. In fact, let's just say there isn't a physics system at all in this game engine. The game has a very specific **state**: each object has a very specific position and velocity and they never change. So imagine this game has a few boxes in the scene (and a camera). In this game, the graphics system is constantly redrawing the scene (even though nothing is moving) at perhaps 100 times per second, just like a movie would show many *frames*, even though several frames are identical on the movie reel at the theater. In this example, this is commonly referred to as "100 FPS" (frames per second). Regardless of the motion of the objects, the image is re-displayed many times per second. In this example, the game has exactly one state.

Now, if we add a physics system to our game, we take this *simulation* and essentially allow multiple **states** plus the ability to **transition from state to state**. The transition from one state to the next state is commonly called an **Update** (or **Physics Update**). The state itself is called a **frame**.

The physics system is what transitions our entire game from state to state. For example, if an object is moving from left to right, it will have the following states: in state 1, object has position (0,0,0), and in state 2, it has position (1,0,0), and in state 3 it has position (2,0,0). This is simplified, but really, when I'm talking about *state*, I'm talking about the game object's position, velocity, orientation, and other properties that make up the game object.

One thing you're going to want to grasp is the concept that the game, as a whole, has a state. Also, as I've mentioned several times before, each game object also has a state. The distinction is that the game state, is composed of every game object's individual state (Figure 1).

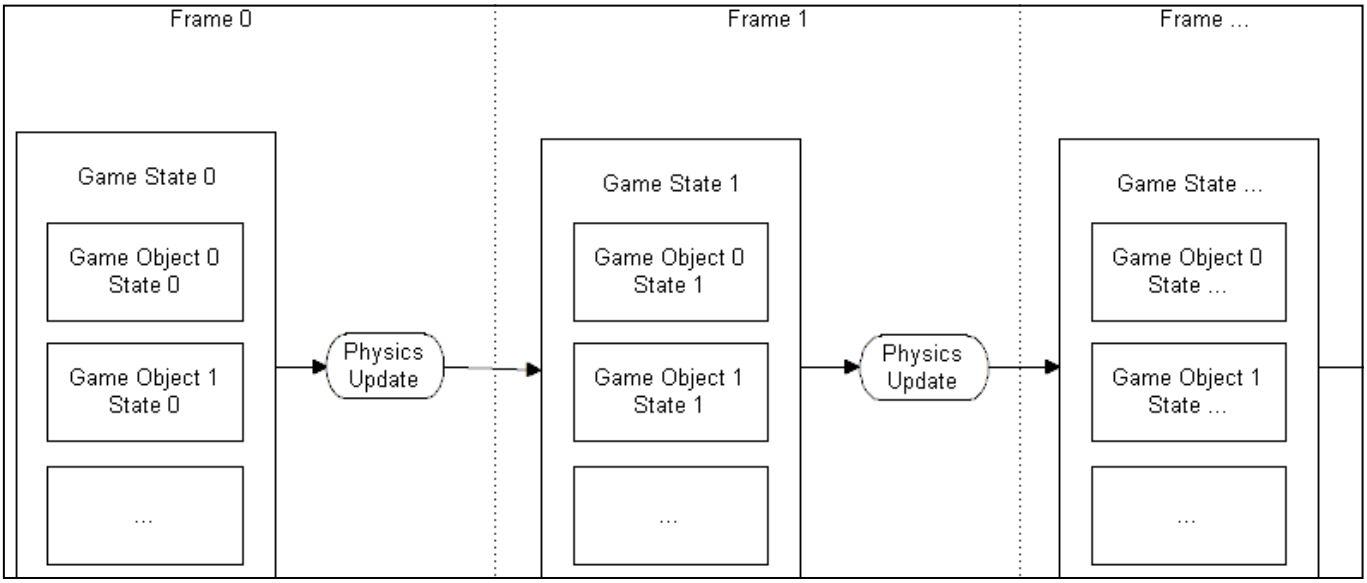


Figure 1: *Game states*

At the end of each Physics Update(), the entire game has a new game state and thus we are on the next *frame*. But, as you can see, it's slightly confusing when everyone throws around the acronym *FPS* like candy at a parade. FPS in the graphics system is not the same as FPS in physics. You can have 100s of graphics frame updates that redraw the scene, but in that same amount of real-world time, you may only have 25 physics frame updates that have happened (state changes).

One interesting thing to note is that we can go from state 0 to state 1, to state 2, etc. Or we can go from state 0 to state -1, to state -2. So we can make the simulation go in reverse if we wanted. (Think of a replay system where you can rewind the simulation.)

Another interesting thing is that, in more advanced simulations, you can update different game objects at different rates. A bird flying around in the background may only need one physics update per second, whereas a race car will need to be updated a lot more often. For the objects that get updated less frequently, you can simply extrapolate their position and orientation. So you do update each object's position and orientation, but you just don't recalculate any of the other physical attributes, such as acceleration and velocities. We discuss how this works later in the article.

Time step

What is a time step? Basically, it's how much time we want our simulation to move forward. Time steps can be **fixed** or **variable** and each has its pros and cons.

Variable

This is the easiest time step to implement. When the time step is equal to the same amount of real-world time that has passed since the last update, it is called a variable time step because the amount of time changes from frame to frame. So the length of the time step is equal to the amount of real-world time in which the previous frame existed. For example, if we are updating frame #5, then we use the amount of time that frame #4 existed (time elapsed since the last update).

Here is a simple example of using the time step to update the position of a game object:

```
position_new = old_position
+ velocity * time_step;
```

Velocity is in (usually) meters per second. If we did one update per second, our equation would look like this:

```
position_new = old_position
+ velocity * 1.0f;
```

Fortunately, we don't do one update per second because, if we did, the object would appear to hop along its path instead of moving smoothly along its path. The `time_step` is scaling our velocity to something much smaller (since `time_step` is usually much smaller than one second.) This equation basically says, *"Add a tiny velocity to the old position and that is our new position."* Since this equation gets run over and over many times a second, it updates the object from position to position in very small increments giving the appearance of smooth motion, just like a movie reel would do when we watch a movie. We don't actually see motion, we see lots of images in rapid succession where the changes are small. This gives the appearance of smooth motion since our brain is, for the most part, unable to distinguish between the two, unless the frame rate is very low (< 12 FPS).

Fixed

A fixed time step is just that, a number that doesn't change. The time step can be large or small. I find that 0.02f is a decent starting point. Both large and small timesteps have pros and cons. Large time steps are faster (CPU computation-wise), but results in a less accurate simulation. Small time steps are just the opposite (computationally expensive but more accurate). Large time steps can also cause issues with springs and collision detection.

So why would you want a fixed update? For starters, it makes the simulation reproducible. Because the time step is exactly the same every frame, you should get the exact position of the objects every time you rerun the simulation with the same starting conditions. Without any outside influences, such as user input, re-running the game should produce the same *solution* over and over again. This is useful for debugging since you could capture the initial conditions (game state), and simply resume the simulation from that point forward. The results should be the same every time.

Using a fixed update also gives you greater control over the timing of the simulation. If you wanted to implement slow-mo, it's a matter of adjusting the time step. The difference is that you don't want the simulation to sync back up with real world time (to catch up to it).

One very important thing to note is that a fixed time step requires you to synchronize the game simulation with the real world time, otherwise your simulation won't run at the correct speed. For example, let's say the simulation started at time 0.00 seconds and we did 6 physics updates to bring the simulation to 3.00 seconds, and the amount of real-world time that has passed is 3.25 seconds, since we started the simulation (Figure 2).

If we did another update, our simulation would be at 3.50 seconds. This is too far, since the real-world time is 3.25 seconds. One solution is to shave off the *0.25* seconds for the next time the engine tells the physics system to update the state.

In order for us to move the simulation forward by 0.5 seconds (run another physics update), we need to make sure that *K* (0.25) is greater than or equal to our time step of 0.5 (J).

Since it's not, we don't do any more updating, we drop out of the physics update loop, and let the engine take control again. When the engine gives us back control, in the physics update, we will probably have something that looks like this: (Figure 3).

Now *K* is greater than our time step (0.5), because the engine took so long to give us back control, so we can move the simulation forward by 0.5 seconds. We again, will have a little left over (0.15 seconds = 3.65 - 3.5). So we drop out of the physics update loop, and we keep track of what's left over until the engine gives us back control again.

It's important to account for ALL the real-world time and to use that time to drive the simulation; otherwise the simulation won't run at the correct speed.

Physics Update

The physics update loop is extremely important in a physics simulation because this is the mechanism which synchronizes the game simulation with the real-world time. I discussed how it works already, but I didn't show any pseudo code for it. This isn't bullet proof, but it should generally get you going in the right direction (Listing 1).

The important thing is that, for a fixed time step, you may need to do multiple physics updates to catch the simulation up to the real world time. For a variable time step, this code looks different and is more simple. For instance, there is no need for a loop in the physics update.

Inertia Tensor

I'm going to sidetrack briefly to discuss the inertia tensor. The inertia tensor is a symmetric 3x3 matrix that is the rotational equivalent of mass. Let's explore this idea. We should all be familiar with the equation F=ma. This is the linear force equation that we learned in basic physics: Force = mass * acceleration. The rotational equivalent to F=ma is $\tau = I\alpha$. Torque = inertia tensor * angular acceleration.

Listing 1

```
void GameLoop( void )
{
    while( true )
    {
        UpdateInput();
        UpdatePhysics();
        UpdateGraphics();
    }
}

void UpdatePhysics( void )
{
    oldStartOfFrameTime = startOfFrametime;
    startOfFrameTime = currentRealWorldTime;
    howLongLastFrameTook += startOfFrameTime -
oldStartOfFrameTime;

    // Keep updating our objects until we have caught our
simulation
    // up to the real-world time.
    while( howLongLastFrameTook >= timeStep )
    {
        // Sometimes our CPU is too slow to catch our
        // simulation up to real world time. This drops out
just
        // in case. Otherwise it will hang forever trying to
        // catch up. I set it here for 3 seconds.
        if( howLongLastFrameTook > 3.0f )
            howLongLastFrameTook = timeStep;

      UpdateAllGameObjects( timeStep );
     howLongLastFrameTook -= timeStep;
}
```

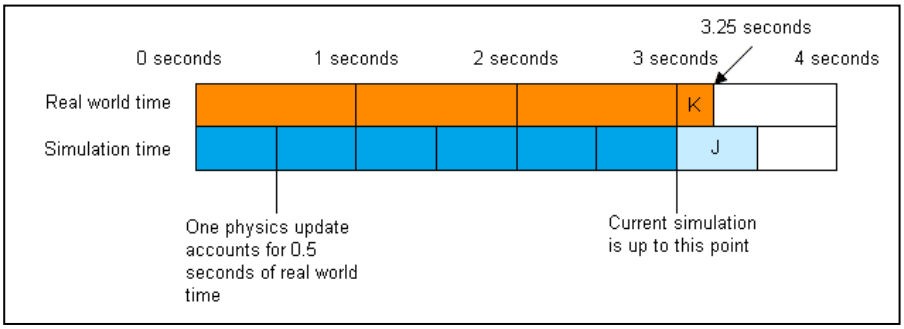


Figure 2: *Time sinchronization*

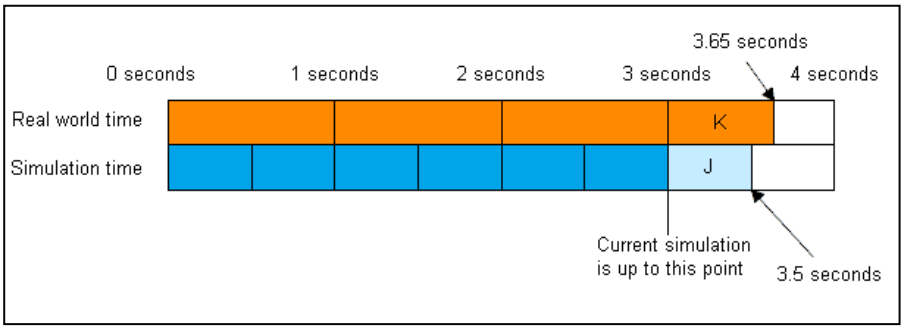


Figure 3: *Time sinchronization II*

When we describe the motion of an object, we know that it has both linear and rotational components. Just like an object can move in three dimensions, an object can also rotate in three dimensions as well. $\tau = I\alpha$ describes this rotational movement in the same way that F=ma describes the linear movement.

Let's do some dimensional analysis first:

Torque = 3D Vector
Inertia Tensor = 3x3 Matrix
Angular Acceleration = 3D Vector

A 3x3 matrix times a 3D vector yields a 3D vector. Fine. Sounds reasonable.

So what is this inertia tensor (3x3 matrix), really? It's essentially the *rotational mass*. The tensor describes the distribution of mass throughout an object. Calculating it by hand gives you a sense of how it works and what it means, but requires triple integrals and density to muck with. But the tensor tells us how much resistance of rotation is around each axis. So, if you want to calculate angular acceleration then you need this inertia tensor. So how do we get it, or calculate it?

You can calculate the inertia tensor on the fly using the volume of the object and the density, but using the inertia tensor of a simple cube for non-cube objects sometimes works reasonably well. The inertia tensor for a solid cuboid is the following:

$$I = \begin{bmatrix} \frac{1}{12}m(d^2+h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+h^2) & 0 \\ 0 & 0 & \frac{1}{12}m(d^2+w^2) \end{bmatrix}$$

Where *d* is depth, *h* is height, *w* is width, and *m* is mass. Note this inertia tensor is defined in the object's local space.

One thing that absolutely cannot be ignored is which space you're currently messing with. Usually the inertia tensor is defined in the object's local space. If you want to compute the angular acceleration from a torque (which I'll discuss later), you need to make sure you do it in the correct space, otherwise weird things will happen. So how do you do this? Well, you will probably have the torque in world space since it's calculated from collision between other objects, so you probably just want to create an "inverse world space inertia tensor" and just keep the torque in world space. It is done by doing the following:

$$I^{-1}_{global} = MI^{-1}_{local}M^{-1}$$

I^{-1}_{global} is the inertia tensor in global space.
M is the model to world transform matrix.
I^{-1}_{local} is the inertia tensor in local space.
M^{-1} is the transpose of the model to world matrix.

What is this equation doing? If we take a vector v, and multiply it through our concatenated matrix set, we get

$$vNew_{world} = [M][I^{-1}_{local}][M^{-1}][v_{world}]$$

We're essentially taking vworld from world space into local space of the object, M^{-1}, then multiplying by the local space inverse inertia tensor, I^{-1}_{local}, and then bringing it back out into global space, M.

We want the inverse because it's required to compute the angular acceleration, derived later in the article. You could also bring the torque into local space, multiply it with the tensor, then bring it back out into world space (which is essentially what this is doing). But this pre-computes that part of it. Keep in mind that all positions, velocities and accelerations are usually in world space anyway.

The Integrator

One way to go about updating the state of every game object is to use an integrator. For simple simulations, you can use closed-form solutions, but for general simulations, closed form solutions rarely exist. We use an integrator to approximate the closed form solution. It's not as scary as it sounds, but essentially an integrator is a finite difference method. Most finite difference methods are derived using the Taylor series expansion. We truncate the expansion, thus introducing a certain amount of error, but the tradeoff is speed.

Usually we will have a force (say, gravity) that we want to affect our game object by. This force uses an acceleration, a, of 9.8 m/s/s. So we have *F = m* * 9.8f;

The problem here lies in the fact that we can only calculate a state change once every so often (an update). If it's constantly being applied to an object (since gravity always is), we need to account for the time between state changes. What happens to the force between state changes? For gravity, this is easy since it's always the same, but for other forces, it's not. The amount of force being applied can change very rapidly.

So how do we calculate the new position of an object knowing this information? Well, we estimate what it will be when we update by assuming the force was constant throughout the last frame, even though we know it may be changing. Since the update should be pretty quick, it seems to be a reasonable assumption. We just need to make sure that we recalculate the forces every frame before we do an update since they may not be the same. We use these new forces and assume they're constant throughout the frame. We don't really have to worry about gravity though because we know it's constant all the time.

What we need is to sum all of these forces being applied for a certain length of time. This amount of

time is of length `time _ step`, since this is the amount of time between state changes.

The things we should have in our physics component for this example:

1. mass (or inverse mass)
2. position
3. velocity
4. acceleration
5. Inertia Tensor (or inverse inertia tensor in local space)
6. Inverse Inertia Tensor in World Space
7. orientation (quaternion)
8. angular velocity
9. angular acceleration

We need to update all of these properties every physics update, (except mass or the inertia tensor, usually). So how do we do this?

We have an equation at our disposal that links force and acceleration:

$$F = ma$$

$$F\frac{1}{m} = a$$

Now that we have acceleration, we can update everything else, and we can do it because we have both the force and the mass of the object. The force in this equation is the sum of all forces acting on it. For example, gravity and wind added together. Since they're just 3D vectors we just add them together.

From this acceleration we calculate our new velocity:

```
vNew = vOld + aNew * time _ step;
```

From our new velocity, we calculate our new position:

```
pNew = pOld + vNew * time _ step;
```

We want to recalculate these values every physics update, and we need to do the same for the rotational equivalents. For the rotational counterpart we have $\tau = I\alpha$.

Now we need to change this torque into an angular acceleration. Solving for alpha in the equation $\tau = I\alpha$, we get $I^{-1}\tau = \alpha$. Now we have a similar looking equation compared to the linear one. Now that we know how to get accelerations from forces, we just need to use the integrator to actually compute them.

The integrator is the piece of code that actually performs the physics state change for each game object (update). You have a lot of options here, but each type of integrator has its trade offs. Probably the easiest integrator to comprehend is the explicit euler: (Listing 2).

The explicit euler integrator is simple and fast, but it's not that accurate. For example, when you deal with springs and decently large time steps, you usually have to intentionally dampen the springs to prevent them from exploding all over the screen. The problem is that it overestimates and ends up adding in more energy into the system than what's supposed to be there. You can use something more accurate such as the Runge-Kutta4 integrator. It is a lot more accurate, but the trade off is that it's expensive.

In this integrator example, I intentionally left out torque computations involving $\vec{r} \times \vec{f}$.

This is because turning a linear force into a torque isn't done in the integrator; it's done in the collision detection and resolution areas of the engine. Collision detection and resolution aren't necessary for a simple physics simulation though, but it sure makes things a whole lot more interesting.

Once you have a simple physics simulation involving point particles, you can then move on to rigid bodies, such as boxes and cubes. Get that running with gravity and perhaps other forces, such as wind, and buoyancy, and then work on implementing a simple collision detection and resolution system. Start with something simple, such as the Separating Axis Theorem. Then you might want to add in friction. Collision detection is probably the most complicated thing in a physics system.

Listing 2

```
void PhysicsUpdate(){
  for_each( gameObject in GameObjects[] ){
    Integrate( gameObject );
  }
}

// The explicit euler integrator
void Integrate( float time_step ){
  // Recompute our Inverse Inertia tensor in world space
  iitws = [modelWorldMatrix][inverseInertiaTensor]
[invModelWorldMatrix];

  // Linear
  acceleration_new = 1/m * force_total;
  velocity_new = old_velocity + acceleration_new * time_step;
  position_new = old_position + velocity_new * time_step;

  // Rotational equivalents...
  angular_acc = iitws * torque_total;
  rot_vel_new = old_rot_vel + angular_acc * time_step;
  orientation_new = old_orientation + rot_vel_new * time_
step;
}
```

Todd Seiler

Todd graduated with a B.S. from DigiPen Institute of Technology and is currently at employed at Game Circus LLC in Dallas, TX as a game programmer where he works on various mobile games for iPhone and Android devices.

VIVID GAMES
EXCELLENCE IN DIGITAL

Vivid Games has earned a robust reputation throughout the industry as a reliable employer that offers a thriving, creative environment for anyone who is passionate about making – and playing – games. A richly diverse catalogue of titles is always on hand to ignite both your imagination and your career. Joining a team at Vivid Games opens the door to an exciting and challenging vocation that can change your life profoundly.

WE'RE LOOKING FOR

- Game Programmers,
- Testers,
- 2D Graphic Artists,
- Producers,
- Game Designers.

BENEFITS

- Flexible working hours,
- Competitive, stable salaries,
- Free soft drinks, coffee, tea, snacks, etc.,
- Great health care benefits,
- Friendly and creative atmosphere,
- Modern, well-located offices,
- Relocation support,
- Free gym membership (in planning),
- Team building events (Christmas party, milestone parties, etc.).

LOCATION

Bydgoszcz, Łódź

CONTACT / INFO

careers@vividgames.com

SKI JUMPING 2012

Ski Jumping 2012

Take command of the snow-covered slopes as you leap into the exciting world of professional ski jumping. Accelerate down the inrun at dangerously high speeds, battle to keep your balance against the powerful, icy wind, then take flight for as you try to beat the world's best ski jumpers in this invigorating winter sports extravaganza.

A vibrant new addition to Vivid Games' million-selling Ski Jumping franchise, Ski Jumping 2012 now includes quick races, tournaments, custom events and world cups at over 20 different real-life ski jumping venues.

FEATURES

- Full 3D visuals.
- Retina display and A5 enhancements.
- Over 20 real-world ski jumping venues.
- Global online leaderboards.
- Quick Play, World Cup, Tournaments and more.
- Custom events to compete in.
- Three different control methods.
- Player customization.

Vivid Games (www.vividgames.com)

Vivid Games is one of Europe's premier emerging independent studios, with a passion for accessible and engaging gaming at the heart of its philosophy. The company is built on a solid commitment to excellence in all aspects of game development, balancing design, innovation and cross-platform technology. It believes in user-focused playability and a strong awareness of the ever changing trends in modern gaming.

Founded in 2006 in the heart of Poland, Vivid Games is making a valuable contribution to the worldwide games development community. Having already expanded its presence to London, England, the company is also spreading across the Atlantic to San Francisco in the United States.

Since its inception, the release of over 150 titles has seen Vivid Games grow to become a globally recognized force in the realm of mobile and digital games. Often working in long-term partnerships with some of the world's leading game companies, Vivid Games has become renowned for its specialization in cross-platform development, working both with licensed brands and exciting new IPs of its own.

VIVID GAMES. EXCELLENCE IN DIGITAL.

Introduction to 3D Collision Detection

by Ruben Alcaniz

Introduction

Welcome to this brief introduction on how to design the code related with the ever-challenging task of detecting collisions between 3D objects. I would like to emphasize that this article is not particularly technical. For further details we can always resort to our dear, boring books. You can use these solutions in a physics engine or just as a standalone collision detection module to see if multiple objects are colliding with each other. The big difference is that in a physics engine that collision produces a reaction that leads to movements, turns, etc ....

About author

Without a doubt, the least important part of this tutorial is this, but it can be important to know that I am a veteran game programmer who a few years ago wrote his own physics engine (LePhysique, surely the best of this engine is its name) for mobile devices.The engine has been used in several games, some sold and others are not.Originally the engine did not use floating point numbers (it was written with fixed point arithmetic) in order to operate at full performance on ARM processors used on devices such as the Nintendo DS .... I ended up using floating-point numbers for my mental health's sake.... Well, at that point, I decided to jump into iPhone and related devices, where I could afford calculations that are more complex.

Colliders

When I talk about *objects* in this article I always implicitly refer to the them as Colliders.

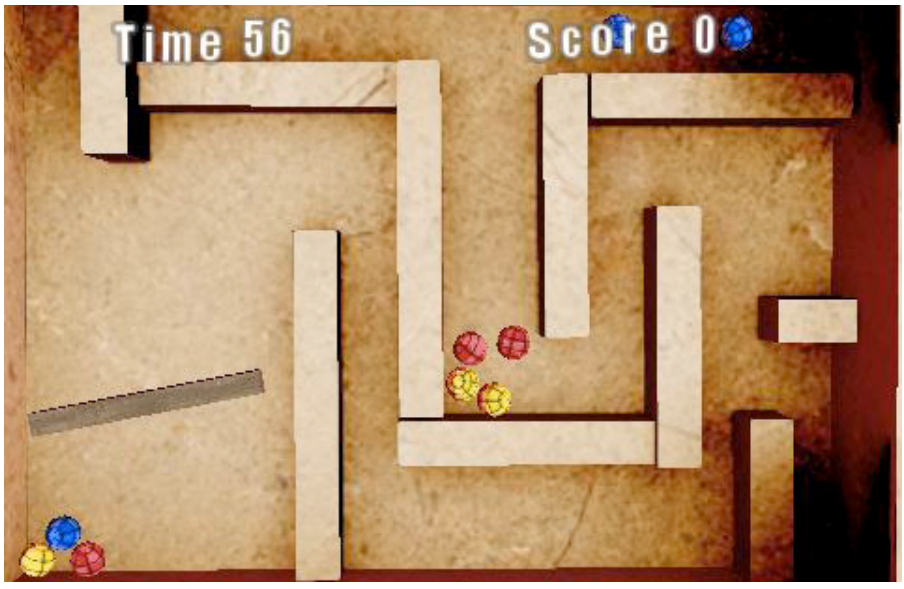


Figure 1: *Spheres Game for iPhone, this game was the first one using LePhysique, you can download, it's free!*

My definition of Collider is: *a geometric shape that is intended to match as close as possible possible to the corresponding visual object it represents with the least complexity possible*. Within the world of object collision detection between objects we highlight the range of primitives that exist:

Spheres, Boxes, Capsules, Cylinders, Convex Mesh...

Unfortunately for us, these primitives have very different approaches to object collision. So you have to have specific code for each pair; for example:

A sphere collides with another sphere.
A sphere hits a box.
A box hits another box.
A convex mesh hits a sphere.

We should write functions that meet all possible cases. OK, if you have girlfriend/boyfriend or any hobbies just forget about it.... Because this is hard work!

An alternate option is to do collision detection with just spheres, box and meshes (not only convex) and dispense with the rest of the primitives for a decent performance. But imagine, for example, a case where you want to have a barrel with its physical behavior ...Ideally you would have support for cylinders or convex meshes, but with a basic approach we could do something like this: (Figure 2).

Coarse collision

To discard potential collisions as fast as possible we need to bind our objects/colliders with a basic physics primitive, which allows us to do a *cheap* check. It is very important to know that when you get the result of two bounding spheres that are colliding, it does not necessarily mean that the objects are actually colliding, because their bounding volumes are always somewhat larger than the objects themselves. When this happens we're talking about a *false positive* impact. In short, we end up doing complex calculations to finally realize that objects have not collided. Life is hard! (Figure 3)

This happens only in some cases, so make sure to choose good *wrappers* (spheres, boxes ...) that are best adapted to the actual shape of the object to avoid as many *false positives* as possible (Figure 4).

Another important issue that we could talk about is collision detection in environments with lots of static objects, using a BVH(Bounding Volume Hierarchy), BSP-Trees, Oct-Trees; this would require another whole article that I would like to write in the future. Basically we will talk about checking objects against large hierarchies represented in trees that allow us to quickly discard areas filled with objects that potentially have no collisions.

Collision detection algorithms

In this chapter we will browse cost methods that come into play when we need to know if two objects collide at some point when their wrappers have collided and we want to know if it will be a "false positive" or a real collision.

It is very common these days to *need* Colliders based in complex geometry. It is always desirable that these shapes are always a simplified versions of the real object, since levels of real-time physics could not afford to check all polygons the mesh *visible* may have.

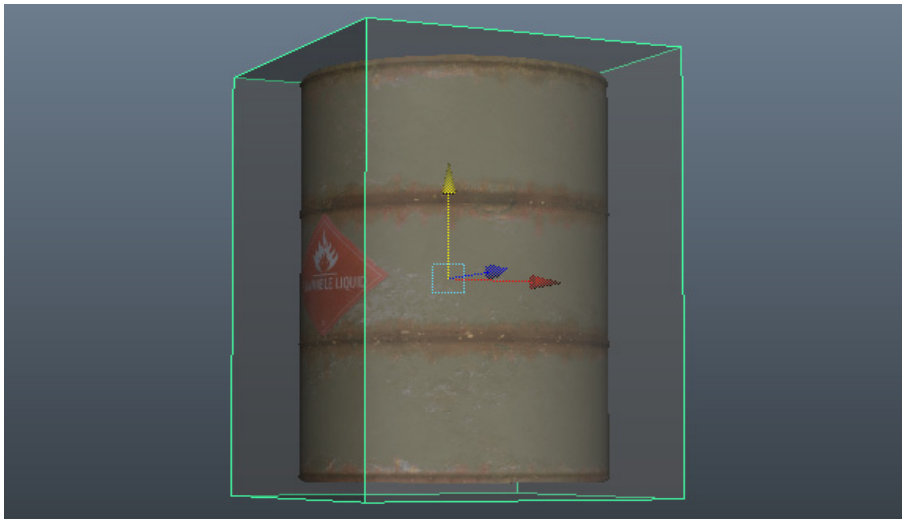


Figure 2: *Box bounding a barrel, the behavior of this object and its collision will not be particularly suitable. But if we cannot afford to have cylinders, this is what we could do. Do not expect to see this object moving very smoothly.*

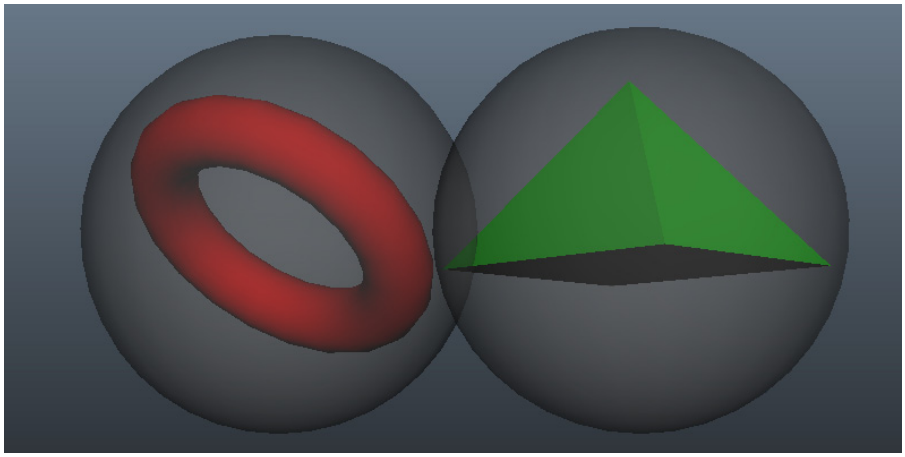


Figure 3: *In the picture we see the gray spheres that represent the basic bounding volumes for a quick test. This example would be a "false positive" because the bounding volumes are in contact but not the objects they contain. (Mr Thorus and Doctor Piramid)*

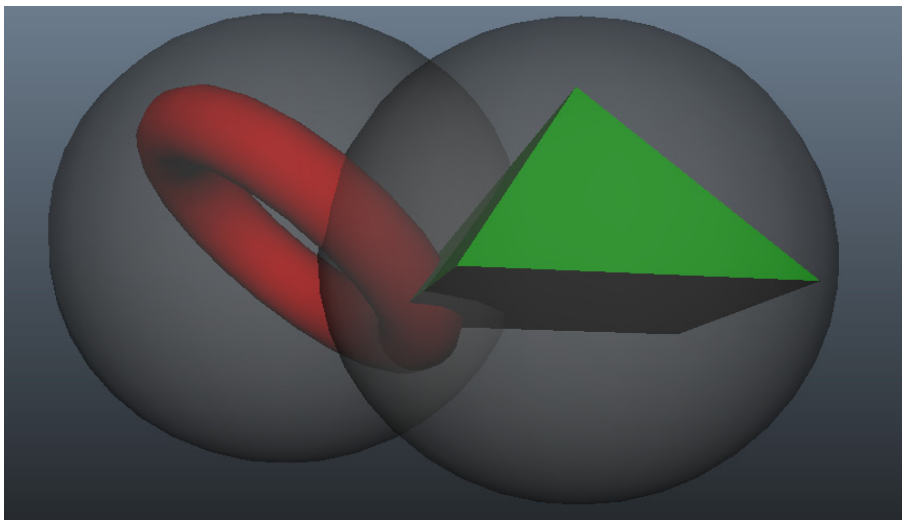


Figure 4: *In this case we clearly see that after checking that the two spheres are in contact, their objects are also in contact. That's what we want to know!*

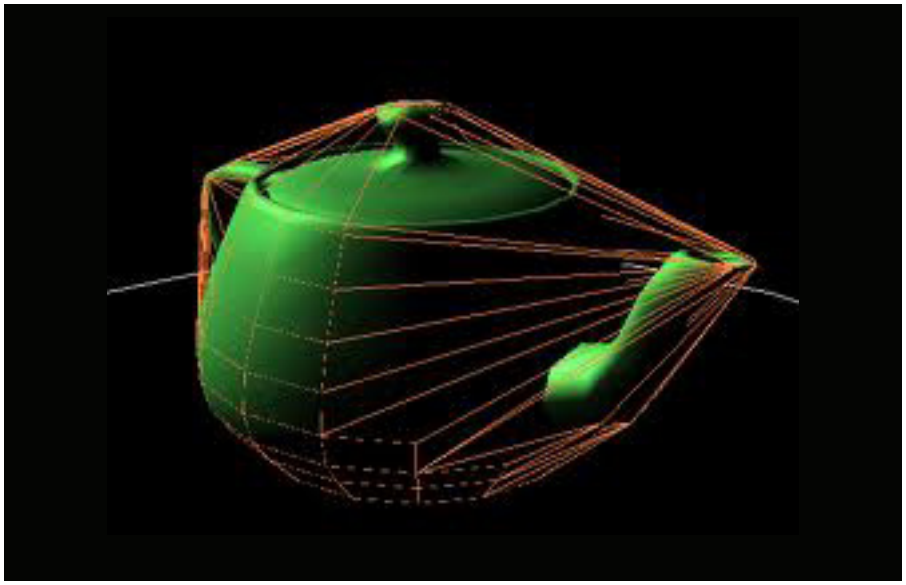


Figure 5: *Famous teapot "surrounded" by convex physical mesh.*

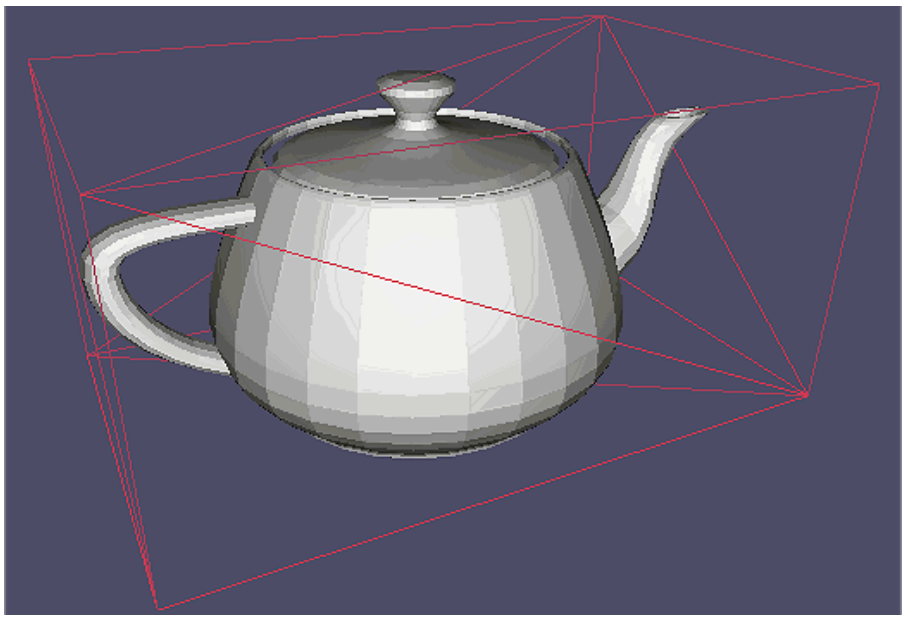


Figure 6: *The same teapot with basic oriented bounding box.*

Furthermore, I recommend using convex meshes for reasonable performance. If you get inside of the world of collision detection algorithms between meshes you will see how much easier convex meshes work (Figure 5).

Sometimes a simple box-oriented (OBB) can be used to wrap objects. Keep in mind that you will have diminished collision accuracy. (Figure 6).

Unfortunately, we cannot review the algorithms here because it would become a book instead of a simple article. In Chapter 7, I provide the references for two books for anyone interested in this topic. When you struggle with testing primitives, keep in mind these cases:

Testing Sphere Against Plane
Testing Box Against Plane
Testing Cone Against Plane
Testing Sphere Against AABB
Testing Sphere Against OBB
Testing Sphere Against Triangle
Testing AABB Against Triangle
Testing Triangle Against Triangle

and

Separating-axis Test

Collision response

As mentioned in the introduction, we will not be discussing the collision responses. Instead, we're only interested in finding out if two or more objects collide with each other efficiently and quickly. For those who want to use these techniques in their own physics engine it is important to note that reaction implementations require more data. These data is usually called Contacts, They store among other items the position of the collision, references to the two colliding and interpenetration. From the contacts, we can determine the reaction of the collision between objects.

Conclusion

The more I analyze the intricacies of decent ..collision detection approaches, the more I ask myself, whether if it is seriously worth it doing from scratch or if using a library would make life easier. I suppose if you're reading this is because you are interested in the subject and I will not stop you. I conclude that the world of 3D collisions is complex. I do not know if you find this brief introduction useful. I wrote it with all my love and I would feel lucky if someone wrote a few lines to my email *ruben.alcaniz@astute-games.com* and tell me that it has been useful to read this, it really would make me very happy. I think there will be dozens of things that are not clear, you can also write me and ask what you want.

References

I will just confine myself to two fabulous books. I remember how expensive they were but it was worth it, I solved a lot of problems. I am sure these books can guide you in collision detection and much more.

Real-Time Collision Detection
The Morgan Kaufmann Series in Interactive 3D Technology

Game Physics Engine Development
Ian Millington

WORLD OF TANKS

World of Tanks is a team-based massively multiplayer online action game dedicated to armored warfare where you can throw yourself into the epic tank battles of World War II with other steel cowboys all over the world. Your arsenal includes more than 150 armored vehicles from America, Germany, France and the Soviet Union, carefully detailed with historical accuracy.

PLAY FOR FREE NOW

GamePro 100/100 "If you've ever had dreams of being a wartime tank commander, you just absolutely have to try it out."
GamingNexus 100/100 "Unique and confident in its place in the free-to-play world, this team-based MMO is a surefire winner, balancing fun with depth and detail with action."
Bit-Gamer 95/100 "It's accessible enough that you can master the controls in seconds, but detailed enough to satisfy hardcore tankheads."

WARGAMING.NET

worldoftanks.com

Definition by Aggregation: An Introduction to the Entity-Component Model for Games Development

by Christopher Mann

What you should know...

- Intermediate knowledge of object orientation
- Basics of vector and matrix math
- Basics of Microsoft's XNA framework

What you will learn...

- How to begin writing your own Entity-Component game engine

Suggested Level of Difficulty: 2

Abstract

The Entity-Component Model is an emergent architecture for developing the back-bone of a game engine. In essence, the Entity-Component Model (ECM) is an attempt to minimise the effort involved when refactoring code (to meet changes in the game design) and maintaining that code for future projects. This article will give a brief outline of how to develop an ECM game engine in C# using the XNA framework.

Indie developers are frequently pushing the envelope of game design beyond that of their triple-A counterparts. Developing an engine that will service such varied projects successfully can be a challenge and minor changes in the design of a game can have engine-wide repercussions in terms of code refactoring.

The biggest benefit of using an ECM is how it defines the objects (or *entities*) that exist in the game world, such as lights, vehicles, weapons and characters. The Entity-Com-ponent Model attempts to abstract itself from genre and content, instead providing the developer with the components needed to create most types of entities easily and quickly.

Previously, the industry trend has been to define these objects by inheritance, where each object in the game inherits from a parent object, which—in turn—inherits from a parent object, and so on until the "root" object is met (Listing 1, ***Annot***. captioned "'Definition by Inher-itance.' Orange nodes are 'concrete' classes that can be added to a game world"). The ECM defines objects not by inheritance, as we are so used to in object-oriented programming, but by *aggregation* (***Annot***. footnote: "Aggregation: the collecting of units or parts into a mass or whole").

The difference between definition by inheritance and definition by aggregation can be simplified: inheritance is saying that something "is a" something else (car *is a* vehicle, vehicle *is a* rigidbody), aggregation is saying that something "has a" collection of things (car *has a* rigidbody, car *has a* model, car *has a* sound emitter, etc.). This means that object are not defined by what methods and properties they inherit, but are defined by the components that they have. We are saying that an entity *is* what it is *made up of* (Listing 2, ***Annot***. captioned "'Definition by Aggregation.' Orange nodes are components, black nodes are entities.").

So, what are the basics of such a system? The first thing we need is an entity class. The entity class will be a single object which will collect components; these components will define it.

Begin by creating an XNA "Windows Game Library" project. We'll now write our first class: the Entity class (Listing 1).

Now we have an Entity class, and we can attach components to it. If you have worked with 3D programming before, you will know that we can manipulate objects in 3D space using Matrix Transformations. In an inheritance hierarchy, the code that enables us to perform these manipulations (scale, rotation, translation) will be in a class very close to the root of the hierarchy. That's because quite a lot of game objects (in fact, everything that appears in 3D space) will need to know where it is, how it is positioned, and the scaling factor applied. You may have noticed that the Entity class has a reference to a Transform: in the ECM, we simply say that *all* entities have a transform (even though some entities might not need a transform). So, we shall continue by defining the Transform class (Listing 2).

Looking at the transform class, we can see that the process for returning a world matrix if the entity is a child is slightly different. This is known as a matrix chain. In essence, a matrix-chain allows us chain our objects together. Matrix-chains are one of the basic pieces of a scene-graph. If you're not sure what the matrix-chain is actually doing, you should see before the end of this article.

Each entity can be manipulated using its transform object. Now that our entities have a position in space, we can write the final piece of the basic ECM: the EntityComponent class. An EntityComponent is a single-service object it does only one job. This must as general a job as possible. Components must be reusable, so it's pointless creating a component that does a very specific job. To keep components similar, we'll use an abstract class from which we can derive any component we need; that way, we can store all of our components in our component dictionary (Listing 3).

It's a very simple class, and it merely says "if you want to be a component, you have to define these methods." The EntityComponent class also contains a reference to the entity to which it has been added, and that helps us access the entity's transform object. That's good, because we'll need it now, as we write our first component: the ModelRenderer (Listing 4).

We have everything we need, except for one thing: we are still blind! We don't have a camera. If we create the camera as a component, then we can attach it to an entity and position the entity anywhere we want.

Finally, let's write the Camera class (Listing 5).

Nothing new here, just a plain old camera. The view matrix that the camera returns takes into account the rotation and position of the entity's transform. So we can just rotate and translate the entity to move the camera around. It also means that we can attach the camera to any entity we want, meaning it will move and rotate with that entity.

The reason we started a game library project is so that we can build our ECM into a .dll, and use it in any game project we want. If you haven't done so, build the solution now.

It's time to test out our library. Start a new project, this time a "Windows Game" project. Add the assets provided to the XNA content project (base, turret, barrel, challenger texture). Now, rewrite your Game1.cs file to read as follows (Listing 6).

We've just made a (very) simple entity-component model. The ModelRenderer will accept any geometry file supported by XNA, so we can build a 3D world quite easily using the tools we've made.

Improvements

Here are some ways to improve your ECM:

Make more components

Very simply, create more components. Create point lights, rigidbody components that give entities physics behaviour (either an interface to a physics engine or your own class), 3D sound emitters, trigger boxes, etc.

Create a messaging system

As we create more components, some components need to pass information between themselves in order to function (for example, a point light must communicate with a model renderer if it is to produce shadows). In order to keep these components "loosely coupled" (i.e. not dependant on one another) we can write a messaging system that carry messages back and forth. This is more desirable than storing references to other components within a component.

Create an Entity Manager

You may want to create an Entity Manager that will perform operations on your entities; operations such as removal and addition from/to the world, finding entities via their string name, and any other manipulations you may need to perform.

To download all the assets please go to: http://dl.dropbox.com/u/14675432/Assets.zip

Christopher Mann

is studying Games Development at Edinburgh Napier University. He is 25 years old and currently finishing his dissertation on the the Entity-Component Model, and will graduate in May '12.

Listing 1: *Entity source file*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace ECMTEST
{
    // using statements and namespace declaration
    public class Entity
    {
        // this dictionary holds any child entities we choose to attach to this entity
        private Dictionary<String, Entity> children = new Dictionary<String, Entity>();

        // the following entity is a reference to the parent of this entity, if there
is a parent
        private Entity parent;

        private Transform transform;

        public Transform Transform
        {
            get { return transform; }
            set { transform = value; }
        }
        // this string is the name of the entity, for "looking up" this entity
        private string name;

        // Follow these properties with some methods or accessors/mutators
        // that will allow the parent and name to be changed. For optimal class
        // de-coupling, this could be achieved through a messaging system

        // for the purposes of demonstration, I have added a mutator for the parent
property
        public Entity Parent
        {
            get { return this.parent; }
            set
            {
                this.parent = value;
            }
        }

        // a simple constructor
        public Entity(string name)
        {
            transform = new Transform(this);
            this.name = name;
        }

        // this method will attach a child entity to this entity
        public void AttachChild(Entity child)
        {
            if (!children.ContainsKey(child.name))
            {
                child.Parent = this; // set the child entity's parent reference to this

                // store the name of the entity, and the entity itself in the
dictionary
                children.Add(child.name, child);
            }
        }
 // this dictionary holds all of the components we choose to attach
        private Dictionary<Type, EntityComponent> components =
        new Dictionary<Type, EntityComponent>();
        // note that the key is of type "Type," this is to make retrieving components
of a particular type faster

        //Very simple component addition method
        public virtual void AddComponent(EntityComponent component)
        {

            if (components.ContainsKey(component.GetType()))
            {
                throw new Exception("Entities can only contain one component of each
type!");
            }
```

```
            components.Add(component.GetType(), component);
            component.OnAdded(this);
        }


        //The following method uses a typed parameter to quickly return the desired
component
        public T GetComponent<T>() where T : EntityComponent
        {
            return components[typeof(T)] as T;
        }

        // when the initialize, update and draw functions are called, pass these calls
onto any attached components
        public void Initialize(GraphicsDevice graphicsDevice)
        {
            for (int i = 0; i < components.Count; i++)
            {
                components.ElementAt(i).Value.Initialize(graphicsDevice);
            }
        }

        public void Update(float deltaTime)
        {
            for (int i = 0; i < components.Count; i++)
            {
                components.ElementAt(i).Value.Update(deltaTime);
            }
        }

        public void Draw(GraphicsDevice graphicsDevice, Matrix viewMatrix, Matrix
projectionMatrix)
        {
            for (int i = 0; i < components.Count; i++)
```

Listing 2: *Transform source code*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;

namespace ECMTEST
{
    // using statements and namespace declaration
    public class Transform
    {
        // the translation of the Transform
        private Vector3 translation;

        // the scale of the Transform
        private Vector3 scale;

        // the rotation of the Transform
        private Matrix rotation;

        // the resultant matrix from the above three factors
        private Matrix worldMatrix;

        // the matrix of the parent entity
        private Matrix parentMatrix;

        // finally, the entity to which this Transform belongs
        private Entity entity;
       // now follows the main method of the transform, GetWorldMatrix, which may be
used when drawing the object
        public Matrix GetWorldMatrix()
        {
            if (this.entity.Parent != null)
            {
                // create three Matrices based on Scale, Rotation and Translation;
multiply those three matrices, in that order.
                worldMatrix = Matrix.CreateScale(scale) * rotation *
Matrix.CreateTranslation(translation);

                // this entity is a child, so we must apply the parent matrix to this
```

```
Transform

                // first we get it
                parentMatrix = this.entity.Parent.Transform.GetWorldMatrix();

                // then we multiply this matrix by the parent matrix, noting the order
                return worldMatrix * parentMatrix;
            }
            else
            {
                // create three Matrices based on Scale, Rotation and Translation;
multiply those three matrices, in that order.
                worldMatrix = Matrix.CreateScale(scale) * rotation *
Matrix.CreateTranslation(translation);

                //this is a top-level entity, and it has no parent, so we can just
return the worldmatrix
                return worldMatrix;
            }
        }
 //Translation is another word for position
        public void Translate(Vector3 translation)
        {
            this.translation += translation;
        }

        //Scale is how big the object is
        public Vector3 Scale
        {
            get { return scale; }
            set { scale = value; }
        }


        //A handy method to reveal which vector is forward, based on the current
rotation
        public Vector3 Forward
        {
            get { return rotation.Forward; }
        }


        //A publicly accessible position property
        public Vector3 Position
        {
            get
            {
                if (entity.Parent!=null)
                {
                    Vector3 v_scale, v_translation;
                    Quaternion q_rotation;
                    GetWorldMatrix().Decompose(out v_scale, out q_rotation, out
v_translation);
                    return v_translation;
                }
                else
                {
                    return translation;
                }
            }
            set
            {
                translation = value;
            }
        }
//Get the rotation matrix from the transform
        public Matrix GetRotationMatrix()
        {
            Vector3 v_scale, v_translation;
            Quaternion q_rotation;
            GetWorldMatrix().Decompose(out v_scale, out q_rotation, out v_translation);

            return Matrix.CreateFromQuaternion(q_rotation);
        }

        //Get the rotation quaternion from the transform
        public Quaternion GetRotationQuat()
```

```
        {
            Vector3 v_scale, v_translation;
            Quaternion q_rotation;
            GetWorldMatrix().Decompose(out v_scale, out q_rotation, out v_translation);

            return q_rotation;
        }

        //Rotate the transform on axis by degrees
        public void Rotate(Vector3 axis, float degrees)
        {
            rotation = Matrix.CreateFromAxisAngle(axis, MathHelper.ToRadians(degrees))
* rotation;
        }

        // if the entity is de-parented—or "orphaned" from its parent it must be
transformed back to world space
        // we may call the following method to do this
        public void TransformIntoWorld()
        {
            Quaternion q_rotation;
            worldMatrix = Matrix.CreateScale(scale) * rotation *
Matrix.CreateTranslation(translation);
            parentMatrix = this.entity.Parent.Transform.GetWorldMatrix();
            Matrix transformedMatrix = parentMatrix * worldMatrix;
            transformedMatrix.Decompose(out scale, out q_rotation, out translation);
            rotation = Matrix.CreateFromQuaternion(q_rotation);
        }
        // this applies the parent matrix one final time, replacing this Transforms
properties with the results
    }

}
```

Listing 3: *EntityComponent source code*

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace ECMTEST
{
    // using statements and namespace declaration
    public abstract class EntityComponent
    {
        protected Entity parent;

private Transform transform;

        public virtual void OnAdded(Entity parent)
        {
            this.parent = parent;
            this.transform = parent.Transform;
        }


        public abstract void Initialize(GraphicsDevice graphicsDevice);
        public abstract void Draw(GraphicsDevice graphicsDevice, Matrix
viewMatrix, Matrix projectionMatrix);
        public abstract void Update(float deltaTime);
        public abstract void LoadContent();

    }
}
```

Listing 4: *ModelRenderer source code*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace ECMTEST
{
    // using statements and namespace declaration
    public class ModelRenderer : EntityComponent
    {
        private Model model;
        private Matrix[] transforms;


        public ModelRenderer(Model model)
        {
            SetModel(model);
        }

        public void SetModel(Model model)
        {
            this.model = model;
            transforms = new Matrix[model.Bones.Count];
        }

        public override void Draw(GraphicsDevice graphicsDevice, Matrix viewMatrix,
Matrix projectionMatrix)
        {
            graphicsDevice.DepthStencilState = DepthStencilState.Default;
            model.CopyAbsoluteBoneTransformsTo(transforms);
            foreach (ModelMesh mesh in model.Meshes)
            {
                // This is where the mesh orientation is set, as well
                // as our camera and projection.
                foreach (BasicEffect effect in mesh.Effects)
                {
                    effect.EnableDefaultLighting();
                    effect.World = transforms[mesh.ParentBone.Index] *
parent.Transform.GetWorldMatrix();
                    effect.View = viewMatrix;
                    effect.Projection = projectionMatrix;
                }
                // Draw the mesh, using the effects set above.
                mesh.Draw();
            }
        }

        //Unused methods inherited from the EntityComponent abstract class
        public override void Update(float deltaTime)
        {
        }

        public override void LoadContent()
        {
        }

        public override void Initialize(GraphicsDevice graphicsDevice)
        {
        }
    }


}
```

Listing 5: *Camera source code*

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
namespace ECMTEST
{
    public class Camera : EntityComponent
    {
        private Matrix projection;
        private float fieldOfView = 70f;
        private float nearClippingPlane = 0.1f;
        private float farClippingPlane = 1000.0f;
        private float aspectRatio;

        public Camera(float aspectRatio)
        {
            this.aspectRatio = aspectRatio;
        }

        public override void OnAdded(Entity parent)
        {

            base.OnAdded(parent);
        }

        public Matrix View
        {
            get
            {
                Vector3 cameraOriginalTarget = new Vector3(0, 0, -1);
                Vector3 cameraRotatedTarget = Vector3.Transform(cameraOriginalTarget,
parent.Transform.GetRotationMatrix());
                Vector3 cameraFinalTarget = parent.Transform.Position +
cameraRotatedTarget;

                Vector3 cameraOriginalUpVector = new Vector3(0, 1, 0);
                Vector3 cameraRotatedUpVector =
Vector3.Transform(cameraOriginalUpVector, parent.Transform.GetRotationMatrix());

                return Matrix.CreateLookAt(parent.Transform.Position,
cameraFinalTarget, cameraRotatedUpVector);
            }
        }

        public Matrix Projection
        {
            get
            {
                return projection;
            }
        }


        public override void Initialize(GraphicsDevice graphicsDevice)
        {

            projection = Matrix.CreatePerspectiveFieldOfView(
                         MathHelper.ToRadians(fieldOfView), aspectRatio,
                         nearClippingPlane, farClippingPlane);
        }

        public override void LoadContent()
        {

        }

        public override void Update(float deltaTime)
        {

        }

        public override void Draw(GraphicsDevice gd, Matrix view, Matrix projection)
        {

        }
    }
}
```

Listing 6: *Game source code*

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace ECMTEST
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;

        Entity wheelbase;
        Entity turret;
        Entity barrel;
        Entity camera;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        protected override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            //This is how we create an entity
            camera = new Entity("Camera");
            wheelbase = new Entity("WheelBase");
            turret = new Entity("Turret");
            barrel = new Entity("Barrel");

            //This is how we add components
            camera.AddComponent(new Camera(16f/9f));
            wheelbase.AddComponent(new ModelRenderer(Content.Load<Model>("base")));
            turret.AddComponent(new ModelRenderer(Content.Load<Model>("turret")));
            barrel.AddComponent(new ModelRenderer(Content.Load<Model>("barrel")));

            //Some intialization for camera graphics
            camera.Initialize(GraphicsDevice);

            //Orienting and positioning the tank pieces until they look tank-like
            camera.Transform.Translate(new Vector3(0f, 40.0f, -150f));
            camera.Transform.Rotate(Vector3.Up, 180f);
            turret.Transform.Translate(new Vector3(0.0f, 12.0f, -5.0f));
            barrel.Transform.Translate(new Vector3(0.0f, -1.0f, 35.0f));

            //If the wheelbase turns, we want the turret to turn also, so we make
turret a child of wheelbase
            wheelbase.AttachChild(turret);

            //Like wise, if the turret rotates the barrel should rotate with it, so
make the barrel a child of the turret
            turret.AttachChild(barrel);

        }

        protected override void UnloadContent()
        {

        }
```

```
        protected override void Update(GameTime gameTime)
        {
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            KeyboardState kbState = Keyboard.GetState();

            //Some tank controls
            if (kbState.IsKeyDown(Keys.W))
            {
                wheelbase.Transform.Translate(-wheelbase.Transform.Forward * 0.5f);
            }

            if (kbState.IsKeyDown(Keys.S))
            {
                wheelbase.Transform.Translate(wheelbase.Transform.Forward * 0.5f);
            }

            if (kbState.IsKeyDown(Keys.A))
            {
                wheelbase.Transform.Rotate(Vector3.Up, 0.5f);
            }

            if (kbState.IsKeyDown(Keys.D))
            {
                wheelbase.Transform.Rotate(Vector3.Up, -0.5f);
            }

            if (kbState.IsKeyDown(Keys.Left))
            {
                turret.Transform.Rotate(Vector3.Up, 0.5f);
            }

            if (kbState.IsKeyDown(Keys.Right))
            {
                turret.Transform.Rotate(Vector3.Up, -0.5f);
            }

            if (kbState.IsKeyDown(Keys.Up))
            {

                barrel.Transform.Rotate(Vector3.Left, 0.5f);
            }

            if (kbState.IsKeyDown(Keys.Down))
            {
                barrel.Transform.Rotate(Vector3.Left, -0.5f);
            }

            base.Update(gameTime);
        }


        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            //Draw the individual entities that make up the tank
            wheelbase.Draw(GraphicsDevice, camera.GetComponent<Camera>().View,
camera.GetComponent<Camera>().Projection);
            turret.Draw(GraphicsDevice, camera.GetComponent<Camera>().View,
camera.GetComponent<Camera>().Projection);
            barrel.Draw(GraphicsDevice, camera.GetComponent<Camera>().View,
camera.GetComponent<Camera>().Projection);

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

Unity 3D

by Gaurav Garg

Unity3d is one of the unique tools, basically used for creating 3D applications such as video games, or other content such as architectural visualizations or real time 3D animations or simulations. Unity's development environment runs on Mac OS X and Microsoft Windows. Applications or games developed in Unity3d are capable for running on the Windows, Mac, Wii, Xbox 360, PS3 and Mobile platforms such as iPhone and Android. There are some system requirements to run unity3d.

System Requirements for Unity Authoring

- Windows : XP SP2 or later; Mac OS X: Intel CPU & "Leopard" 10.5 or later.
- Graphics card with 64 MB of VRAM and pixel shaders or 4 texture units.
- Using Occlusion Culling requires GPU with Occlusion Query support.

System Requirements for Unity iOS Authoring

- An Intel-based Mac.
- Mac OS X "Snow Leopard" 10.6 or later.

System Requirements for Unity Android Authoring

- In addition to the general system requirements for Unity Authoring.
- Windows XP SP2 or later; Mac OS 10.5.8 or later
- Android SDK and Java Development Kit (JDK)

System Requirements for Unity-Authored Content

- Windows 2000 or later; Mac OS X 10.4 or later.
- Pretty much and 3D graphic card, depending on complexity.
- Online games run on all browsers, including IE, Firefox, Safari and Chrome, among others.
- Android authored content requires devices equipped with:
  - Android OS 2.0 or later
  - Device powered by an ARMv7 (Cortex family) CPU
  - GPU support for OpenGLES 2.0 is a recommended.

Unity can also be used for making browser games which require Unity web player plugin. This plugin only supports Mac or Windows but not Linux. Unity also has some major features which can be used for making incredible apps. Unity Provides integrated development environment with hierarchical, visual editing, detailed property inspectors and live game preview. Unity applications can be deployment on multiple platforms as I already mention all the platforms above.

When you load an assets into unity, they are automatically imported and if you update the assets they all are re-imported. Unity also supports bump mapping, reflection mapping, parallax mapping, screen space Ambient Occlusion, dynamic shadows, render-to-texture and full-screen post processing effects. The main power of unity is its Built-in support Nvidia's PhysX physics engine, which helps a lot to make realistic games. Unity also added support for real time cloth on arbitrary and skinned meshes, thick ray casts and collision layers. Unity has supports of Audio system as well as video system. There are few more features which help to make 3d games more interactive such as:

- A terrain with supporting tree billboarding.
- Occlusion culling that helps to increase frame rates.
- Built-in lightmapping and global illuminating.
- Multiplayer networking supports.

There are two phases for making a game in Unity3d, First one is **designing** and last one is **programming**. In terms of Designing, unity supports files created by best 3D applications such as 3dMax, Maya, blender and cinema4D. Unity also supports DXT Textures compression and true type fonts. Hence level designing phase can be achieved in unity3D by creating terrain and using 3D models files and add some special effects using Particle and shaders.

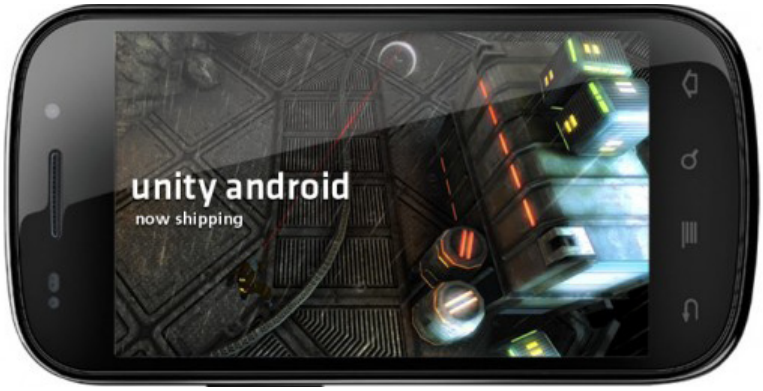


Figure 1: *Unity for android*

In terms of Programming, coding within Unity is done with the unity editor, built on mono, an open source implementation of the .NET framework. The Unity Editor supports coding done in C#, JavaScript, and Boo.

Unity gives two types of licenses: Unity and Unity Pro. Unity Pro version is available for a price and non pro version is free to use. The difference between Unity and Unity Pro are their some features which are available on Unity Pro, such as render-to-texture, occlusion culling, global lighting and post-processing effects. Unity pro has ability to work in a team via version control. On the other hand, the free version displays a splash screen in standalone games and a watermark in web games that cannot be disabled or customized. There are some similarities between unity and unity pro too such as both provides development environment, tutorials, sample projects and content, supports via forum and future updates in the same major version (Figure 1).

Licence of Unity for iOS and Unity for Android are available for purchase separately. A unity pro licenses is required to purchase an iOS pro or Android Pro licenses. The basic Android and iOS licenses can be used with the free version of unity.

The Island Demo

Unity provides an environment which is easy to learn and development. You can play a lot of browser based Unity3d games so you can get an idea of what the engine capabilities and what can engine do. You can download the free version of Unity3d and mess around with the default Island Demo (Figure 2).

You will see the environment like this when you first open the Unity3d. To Play or try out the demo, Click on the play button available at the top-center of the screen ( ).

You can walk around the demo using WASD keys on your keyboards. You can Jump also by pressing Spacebar. Click on the Play button again if you want to end the demo.

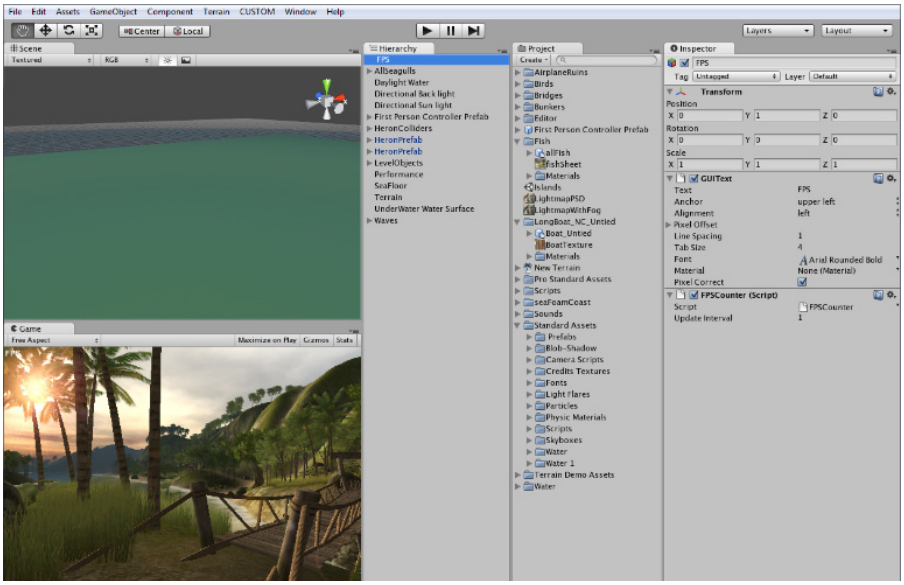


Figure 2: *Defaul island demo*

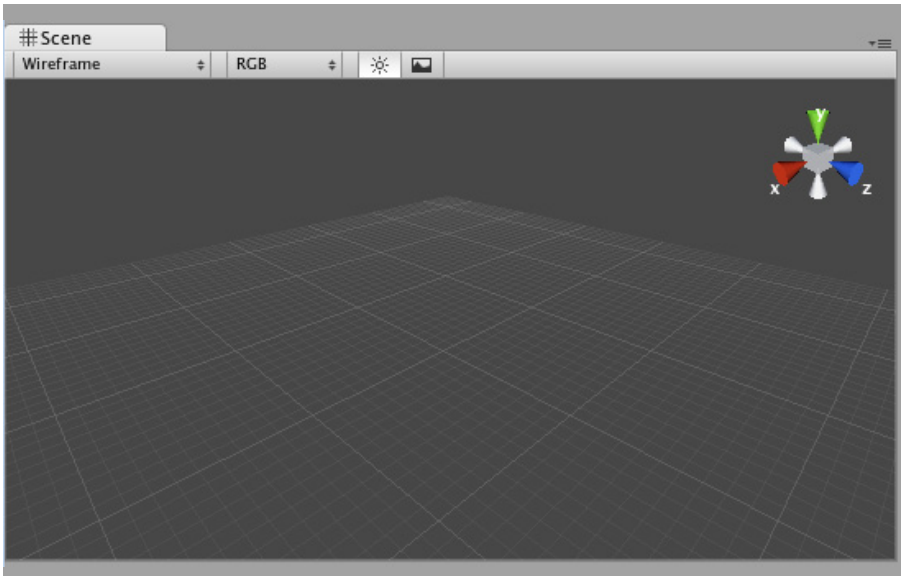


Figure 3: *SceneView*

Figure 4: *Maximize on Play button*

Unity has all the tools to create an island similar to the demo. Unity already made a First Person Controller Prefab object that you can place in your scene or world with WASD keyboard controls that will allow you to move in the terrain.

Let's take a quick tour of User Interface for an Interest. There are few panels I am going to talking about such as Scene window, Game window, Hierarchy, The Inspector and The Project.

The Scene Window

The scene window is where you can adjust and move gameObjects where you want to place. This window has various controls to change its levels of details. We can use these controls to display the contents with texture, wireframe or combination of both, to toggle lighting on and off. There is colourful gizmo in the top right corner to constrain the window to the X, Y and Z axes to view the top and sides of the scene. Click on the white box in the middle to view the scene in perspective view. Your window looks like that when you open a new project or create a new scene (Figure 3).

The Game Window

The Game Window shows the final output what you have done in your scene view. This window show you the final output of the game. You can play the game on this window. Click on the play button to play the game. Toggle the Maximize on Play button to test your game in full-screen mode (Figure 4; Figure 5).

The Hierarchy

The hierarchy panel contain a list of all the gameObjects in scene. GameObject is everything in your scene; it could be a camera, a light, a prefab, and models. These things help to make up a game.

Select a game object in hierarchy panel and then hover the mouse on scene view, if you press F the scene view zoom in that object.

The Project

The project panel consist of all the elements Used for creating game objects. Unity automatically made a default folder called Assets folder when we create a new project. We have to place each and everything inside this folder such as textures, materials, sound files, mesh files, prefabs. By moving or deleting stuffs in Assets folder from operating system will mess up everything in unity project. If you want to change or delete something, do it from inside unity in project panel (Figure 6).

The Inspector

The Inspector panel display the property of a gameobject. When we create an empty gameobject we get default Transform component attach to it. Using Transform component we can rotate, move and scale the gameobject. We can attach different component in a gameobject according to game need (Figure 7).

It was a short summary on Unity3d Interface. Unity3d Interface is designed in such a way that you can customize it. You can write custom scripts to add certain buttons and panels inside unity to speed up the work flow.

This article shows the power of unity3d, what the unity3d can do in easy and better way.

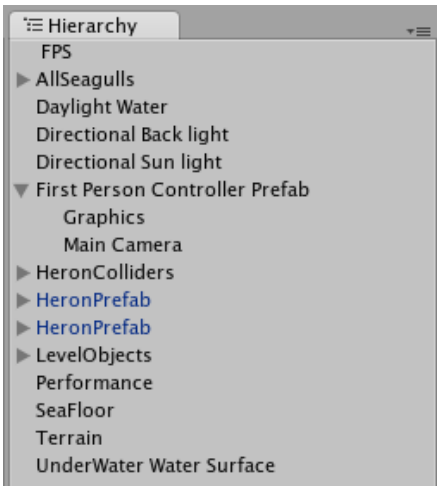


Figure 5: *Hierarchy*

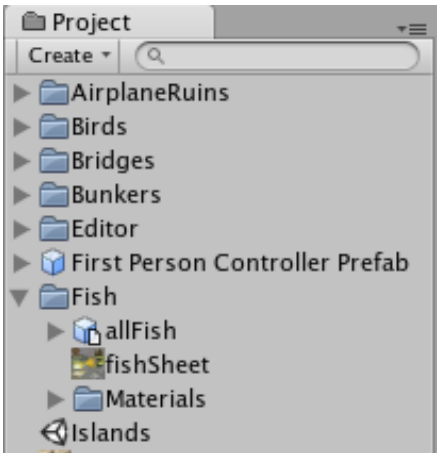


Figure 6: *The project panel*

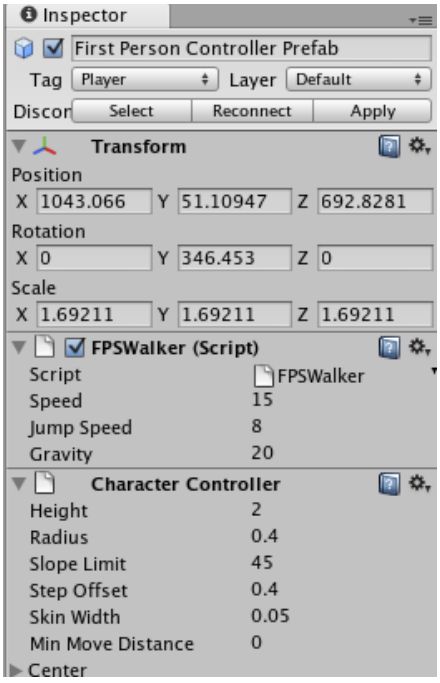


Figure 7: *Inspector*

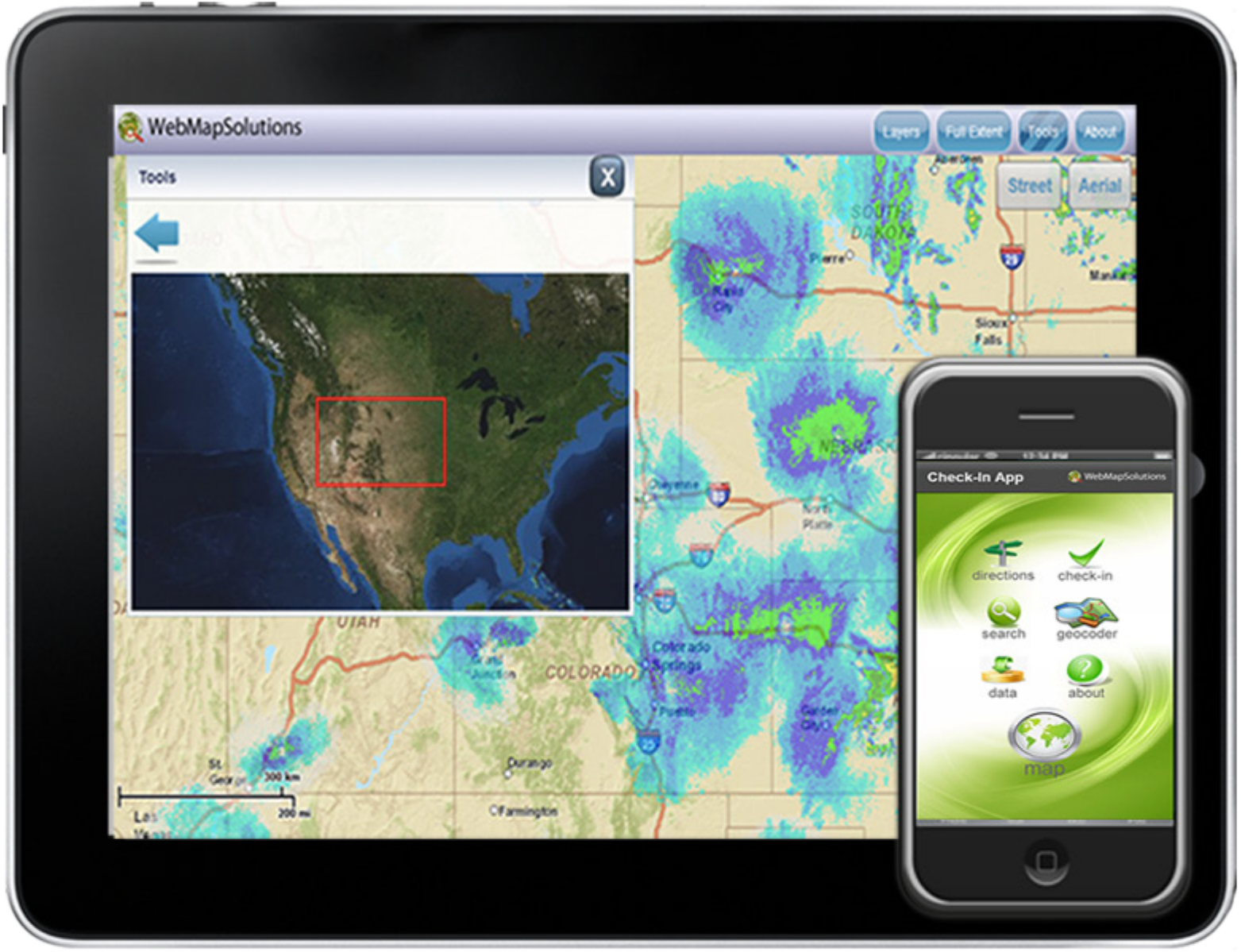
ABOUT ME

Hi everyone, I am **Gaurav Garg**, *Professionally a Game Programmer. Game Programming is not a just work for me; it's what I want in my life. I love to play games also. Apart from Gaming, Photography and reading technical articles are the few in which I like to share my time.*

games@alawar.com

DEVELOPERS WANTED

www.alawar.com



Smartphone or Tablet

iOS, Android, Blackberry, Windows

Let us build your next mobile application...

WebMapSolutions.com

001.801.733.0723

matt@webmapsolutions.com