Table of contents

TimeOfLastIteration
Delay
Interval
TimeUntilNextIteration
DurationLeft
DurationTotal
Duration
IterationsTotal
IterationsLeft

Methods

Cancel
Execute

vp_TimeUtility

Units

Methods

TimeToUnits
UnitsToSeconds
TimeToString
SystemTimeToString
SystemTimeToUnits
SystemTimeToSeconds
TimeToDegrees
SystemTimeToDegrees

Debug Mode

Debug info

Created
Inactive
Active
Show Id
Show Callstack

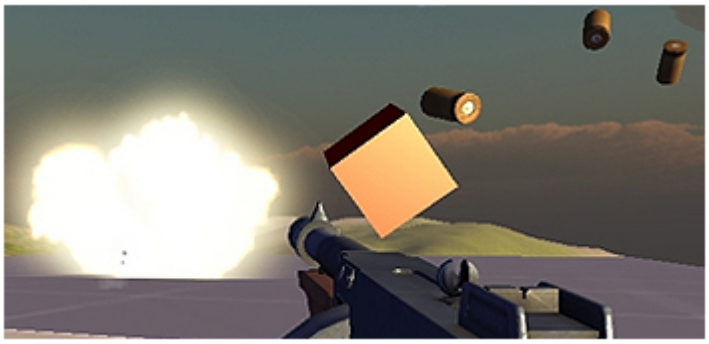
About the object pool

Support and additional information

Introduction

Thank you for purchasing visionTimer! A lot of effort has gone into making this package powerful, easy to use and flexible. Hopefully it will become an invaluable part of your gameplay scripting toolbox.

This system initially grew out of a need for short, direct syntax when doing timing in cutscenes. After making vp_Timer a generic class, I quickly started using it for everything: animated menu systems, gameplay and sound coding (and most recently all the weapon timing in Ultimate FPS Camera).



Ultimate FPS Camera uses vp_Timer for timing weapon switching, firing rate, respawning and explosions, among other things.

For this product package, *vp_Timer* has been rewritten from scratch and a utility class has been added for more of a complete time handling framework. It should now cover most time related cases you're likely to encounter in game programming.

Again, thanks for buying this asset. Feel free to show off your work or participate in the official forum discussion at:
http://forum.unity3d.com/threads/157013-VisionTimer-RELEASED

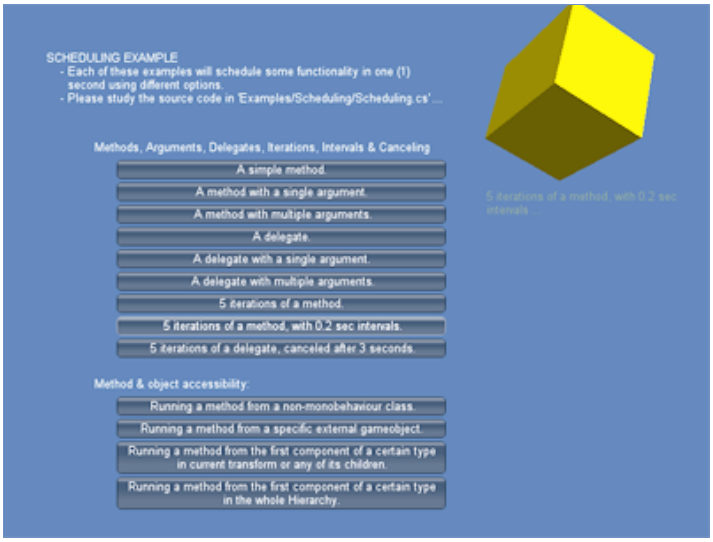
/Cal

Example scenes

visionTimer comes with four example scenes, the purposes of which are to show the system in action and explain usage of the vp_Timer and vp_TimeUtility classes.

Scheduling

This example illustrates a number of common use cases for the vp_Timer class. It covers methods, arguments, delegates, iterations, intervals, canceling and how to call methods on external objects. It is intended as a quick-reference for when you're interested in how to perform a specific task.



StopWatch

If you're developing a racing game, this is the script you will want to start picking apart! The example shows how to set up a GUI stopwatch displaying seconds and hundredths. It uses a vp_Timer object running indefinitely, which can be stopped and paused from script or using buttons. The included prefab can be drag-dropped into a scene and used for a game right away.

Clock

This example shows how to use the vp_TimeUtility methods to set up a traditional GUI clock showing the current system time (it does not use vp_Timer objects, only the utility class). Note that this example renders a clock using the GUI, but you could easily use the same vp_TimeUtility methods for rotating 3D objects such as the hour- and minute hands of a clock tower.

TimeBomb

This is an example of using multiple timer handles to create more advanced animation sequences - in this case for a time bomb display counting down to a pre-blast state before finally exploding. Timer handles are used to blink numbers, to vibrate the display depending on timer duration and to determine when to play various sounds.

Important concepts

If you're like me and skip through manuals, here's what you *really* need to know:

Timers are not Components

visionTimer is a *scripting-operated* system. Though the included example gui prefabs can be added to gameobjects as drag-and drop components, the timers under the hood can not. A timer is always created at runtime, from a static function call inside the script using it.

vp_Timer

vp_Timer is a script extension for delaying (scheduling) game events with a time delay. Timer events are mostly created using the static method In, which has many parameters covering various timing cases.

vp_Timer.Handle

vp_Timer.Handle objects are used to keep track of specific timers and expose a number of useful properties and methods.

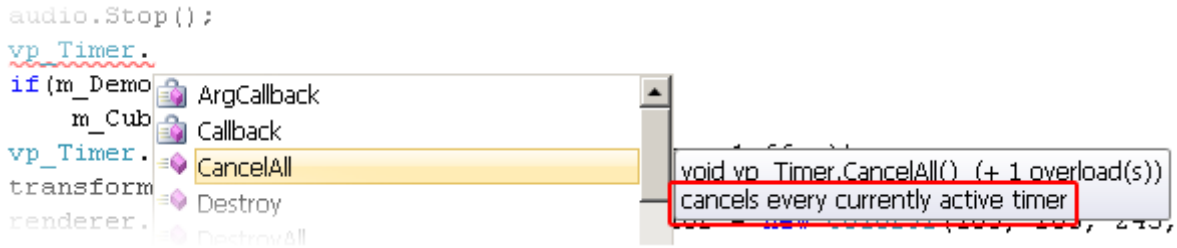
vp_TimeUtility

The vp_TimeUtility class is used for common time-related unit conversions, such as converting an amount of seconds to an angle on a clock face, or converting the current system time into a formatted text string.

Also worth noting

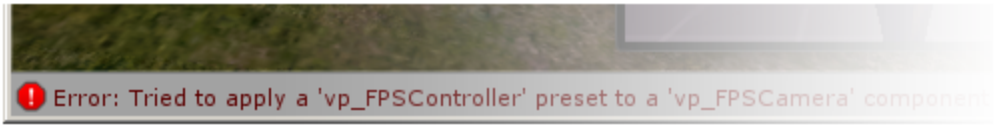
Summary tags

All methods in the included scripts use XML `<summary>` tags. Depending on your IDE (e.g. MonoDevelop / Visual Studio), this means you should be able to mouse-over on a method name in the code (or on a member name in a pop-up list) to display a brief description.



Keep an eye on the Console

Unity communicates any encountered errors and issues via the *Console* (far bottom left in the Editor window). It's always a good idea to be aware of any red (Error) or yellow (Warning) messages it displays. Click on the icon to see the full list of errors. Sometimes the latest error is not the culprit and you need to look further up the list.



What's with the "vp_" filename prefix?

"vp_" is the publisher prefix for *VisionPunk*. It mainly serves to prevent filename collision with scripts from other asset publishers or Unity core classes.

vp_Timer

vp_Timer is a static class for delaying (scheduling) methods with a time delay. It supports arguments, delegates, repetition with intervals, infinite repetition and event canceling. Timer events are mostly created using the static method In.

> ***NOTE:*** *vp_Timer takes a programmatic approach to timing as opposed to content based (using animation curves etc). Though you can animate things using vp_Timer (see the TimeBomb example) it is primarily a tool for timing things rather than an tool for animating them.*

Scheduling events using vp_Timer.In

vp_Timer.In executes a method after a certain delay in seconds. In its simplest form, it's called with a delay as the first argument and a target method as the second.

```
// --- EXAMPLE: execute 'SomeMethod' in one second
vp_Timer.In(1.0f, SomeMethod);

void SomeMethod()
{
	// do something
}
```

- The target method must be of type **void ()**. It can optionally have a single argument of type **object**.
- The **delay** parameter must be zero or positive. If it's zero (0.0f), the event will execute right away.

> ***NOTE:*** *You can start a timer that runs forever without executing anything by calling the vp_Timer.Start method.*

Iterations (repetition)

To repeat an event, simply specify a number of iterations after your method name. This will repeat the method a number of times, separated by a timespan equal to the initial delay.

```
// --- EXAMPLE: execute 'SomeMethod' in one second,
// then repeat it four times with one second inbetween
vp_Timer.In(1.0f, SomeMethod, 5);
```

- To repeat an event indefinitely, specify zero iterations.
- If In is called with iterations but with no interval, the delay will be used as the interval too.

- If you wish to iterate something a number of times but want the first iteration to trigger right away, simply specify a zero delay.

```
vp_Timer.In(0, SomeMethod, 3, 1);
```

Intervals

If you need the delay between iterations to be different from the initial delay, add the desired interval at the end.

```
// --- EXAMPLE: execute 'SomeMethod' in one second,
// then repeat it four times with 0.2 seconds inbetween
vp_Timer.In(1.0f, SomeMethod, 5, 0.2f);
```

> ***NOTE:*** *If you use iterations with a zero delay and do not specify an interval, the interval will also become zero and all iterations will happen simultaneously (this can actually be used as a short-hand alternative to a for-loop if you want to call a method many times in the same frame).*

Arguments

To call a method with a single argument, first create a method of type **void** with a single parameter of type **object**. Then you may call the method with an argument of any type right after the method name. In the target method you must cast the object back into the destination type.

```
// --- EXAMPLE: call a method with a single argument
vp_Timer.In(1.0f, MethodWithSingleArgument, "Hello World!");

void MethodWithSingleArgument(object o)
{

    // use the object directly by type casting it like this ...
    MethodThatTakesAString((string)o);

    // ... or store it as the target type before use like this
    string s = (string)o;
    print(s);

}
```

> ***NOTE:*** *If you pass an integer as the third parameter, it will be interpreted as an argument only if there is a corresponding method that takes an object argument. If not, it will be interpreted as an interval.*

Multiple arguments

Multiple arguments can be sent to a method by passing them in the form of an object array. In this way you can transfer an unlimited amount of arguments of varying types.

```
--- EXAMPLE: call a method with multiple arguments
vp_Timer.In(1.0f, MethodWithMultipleArguments,
                  new object[] {"December", 31, 2012});

void MethodWithMultipleArguments(object o)
{
      object[] arg = (object[])o;  // 'unpack' object back into an array
      print(  "Month: " + (string)arg[0]
             + ", Day:" + (int)arg[1]
             + ", Year:" + (int)arg[2]);
}
```

Delegates

A **delegate** is basically a pointer to a method. vp_Timer uses delegates to reference the methods you specify via the In method. But nothing prevents you from declaring a delegate right inside the In method. This is where delegates become very interesting. They provide the powerful option of wrapping blocks of code inside a timer without having to create a new method every time.

```
// --- EXAMPLE: print a console message in one second using a delegate
vp_Timer.In(1.0f, delegate()    {    print("Hello World!");    });
```

Delegates with arguments

Delegates, too, can be passed arguments in exactly the same way as methods.

```
// --- EXAMPLE: inflict 'incomingDamage' points of damage after a delay
float incomingDamage = 242.0f;

// add a single object parameter inside the delegate's parenthesis
vp_Timer.In(1.0f, delegate(object o) // <--
{
    InflictDamage((float)o);
}, incomingDamage); // <--
// pass the argument right after the delegate's closing bracket
```

Delegates with multiple arguments

Delegates can take multiple arguments in much the same way as methods.

```
// --- EXAMPLE: print the month, day and year variables after a delay
string month = "December";
int day = 31;
int year = 2012;
vp_Timer.In(1.0f, delegate(object o)
{

	object[] arg = (object[])o;   // 'unpack' object into an array

	print(  "Month: " + (string)arg[0]
		+ ", Day:" + (int)arg[1]
		+ ", Year:" + (int)arg[2]);

}, new object[] { month, day, year }); // <--
// pass the month, day and year variables as an object array right
after the delegate's closing bracket
```

Canceling timer events

You can use the **CancelAll** method either to cancel all currently running timers, or only the ones scheduling a specific method. You can also cancel specific active timers using the vp_Timer.Handle.Cancel method (see the next chapter).

```
// --- EXAMPLE 1: cancel every running timer
vp_Timer.CancelAll();

// --- EXAMPLE 2: cancel every running timer that
schedules 'SomeMethod'
vp_Timer.CancelAll("SomeMethod");
```

Running a timer forever using vp_Timer.Start

The **vp_Timer.Start** method can be used to run a timer for the sole purpose of measuring time (useful for e.g. stopwatches). It takes a mandatory vp_Timer.Handle (see the next chapter) as only input argument, has no callback method and will run practically forever. The timer handle can then be used to pause, resume and poll all sorts of info from the timer event

```
// --- EXAMPLE: start a timer that runs forever
private vp_Timer.Handle m_RaceTimer = new vp_Timer.Handle();
..
void BeginRace()
{
	vp_Timer.Start(m_RaceTimer);
}
```

vp_Timer.Handle

vp_Timer events are typically "fire-and-forget". They are very lightweight and can not be manipulated directly. The upside of this is that for most timers you don't have to do a lot of object creation (the downside, ofcourse, being that you can't really access the timer event itself).

vp_Timer.Handle objects are used to keep track of timers post-initiation. Timer handles expose a wide range of useful properties and methods. They are most commonly used to cancel an event or to see if it's still active, but they also have many properties to analyze the state of an event (for example, the editor uses them to display debug info).

Creating timer handles

Timer handles need to be created separately, then initialized by passing a vp_Timer.Handle object as the final argument to vp_Timer.In.

```
// --- EXAMPLE 1: execute 'SomeMethod' in 20 seconds,
// but cancel if user presses space
private vp_Timer.Handle Timer = new vp_Timer.Handle();
..
vp_Timer.In(20.0f, SomeMethod, Timer);
..
if(Input.GetKeyUp(KeyCode.Space))
      Timer.Cancel();

// --- EXAMPLE 2: execute 5 iterations of 'SomeMethod',
// but cancel after 3 seconds
private vp_Timer.Handle Timer = new vp_Timer.Handle();
..
vp_Timer.In(0.0f, SomeMethod, 5, 1, Timer);
vp_Timer.In(3, delegate() { Timer.Cancel(); });
```

> ***TIP:*** *Placing the handle name on the next line of code can sometimes be good way of highlighting the fact that a timer event has a handle associated with it.*

Canceling and blocking timer events

Timer handles allow you to cancel specific timers whenever you like. They can also be used to prevent scheduling a method multiple times simultaneously, or to cancel a running timer event and restart it.

This is all great for GUI and input cases where users may be "spam-clicking" buttons or toggling quickly between screens, player states or weapons. Without this capability, any "toggling system" using timers may fall into very strange patterns of states flipping on and off.

```
// --- EXAMPLE 1: cancel and restart a timer on each subsequent call
private vp_Timer.Handle Timer = new vp_Timer.Handle();
..
vp_Timer.In(0.4f, SomeMethod, Timer); // <--
// if 'Timer' is already active, it will be canceled and restarted

// --- EXAMPLE 2: prevent scheduling a timer event until the
// timer handle we want to use is free (inactive)
private vp_Timer.Handle Timer = new vp_Timer.Handle();
..
if(!Timer.Active) // prevents restarting 'Timer' until it's done
      vp_Timer.In(1.5f, SomeMethod, Timer);
```

Properties

The following properties can be accessed from a timer handle. Please note the difference between properties that return a *timespan* versus those that return a *point in time*.

- A **timespan** refers to an *amount of seconds* that can take place anytime.
- A **point in time** refers to a specific *time since the start of the application*.

Active

Returns true if the timer event of this handle is active (running or paused). Returns false if not, or if this timer handle is no longer valid.

Paused

This boolean can be set to pause or unpause a timer event.

TimeOfInitiation

(point in time) Returns the time of initial scheduling in seconds. That is, the point in time when the call to vp_Timer.In or vp_Timer.Start was made.

TimeOfFirstIteration

(point in time) Returns the time of first execution in seconds. That is, the point in time when the scheduled method will first execute (or was first executed).

TimeOfNextIteration

(point in time) Returns the expected due time of the next iteration of an event in seconds.

TimeOfLastIteration

(point in time) Returns the expected due time of the last iteration of an event in seconds.

Delay

(timespan) Returns the delay before first execution in seconds.

Interval

(timespan) Returns the repeat interval of an event in seconds.

TimeUntilNextIteration

(timespan) Returns the time left until the next iteration of an event in seconds.

DurationLeft

(timespan) Returns the current total time left (all iterations) of an event in seconds.

DurationTotal

(timespan) Returns the total expected duration (due time + all iterations) of an event in seconds. It does not take pausing into account.

Duration

(timespan) Returns the current age of an event (lifetime since initiation) in seconds.

IterationsTotal

(timespan) Returns the total expected amount of iterations of an event upon initiation.

IterationsLeft

Returns the number of iterations left of an event.

Methods

Cancel

Cancels the event associated with this handle, if active. After an event has been canceled, the timer handle still exists. You can poll it for (by now historical) data on the timer event (for example its scheduled total duration) but some properties will be null and you can't make it execute again. You will sometimes want to check to see if the associated event is still running using the Active property.

Execute

Executes the event associated with this handle early, if active. This is useful if you have a scheduled an action that the player should be able to trigger early without canceling it (let's say the player shoots at a time bomb instead of disarming it).

vp_TimeUtility

The vp_TimeUtility class contains a number of static utility methods for performing common time related game programming tasks, mainly converting time spans between different time units.
For example: converting an amount of seconds to an angle on a clock face, or converting the current system time into a formatted text string. It does not depend on the vp_Timer class and can be used independently of it.

Units

The **Units** struct is used by many of the vp_TimeUtility methods to represent a timespan. It holds a time divided into all of the supported standard time components in integer format, namely: **hours**, **minutes**, **seconds**, **deciseconds** (tenths), **centiseconds** (hundredths) and **milliseconds**.

Timespans can also be used to represent times of day, counting from 00:00 hours. For example: the time of day "4:45:29 PM" would convert into a Units struct containing 16 hours, 45 minutes and 29 seconds.

Methods

TimeToUnits

Takes a floating point time in seconds and returns a Units struct representing that time divided into standard units. The value stored in each returned variable represents a fraction in units of the previous larger time component.

UnitsToSeconds

Takes a Units struct and returns a floating point time in seconds.

TimeToString

Takes a floating point time in seconds and returns a string with the time formatted as a configurable list of standard time units, delimited by a char of choice. This is useful for digital time displays such as those typically found in racing games.

SystemTimeToString

Takes a System.DateTime object and returns a string with the time formatted as a configurable list of standard time units, delimited by a char of choice. Omit all parameters to default to the current system time.

SystemTimeToUnits

Takes a System.DateTime object and returns a struct representing that time divided into standard Units. The value stored in each returned variable represents a fraction in units of the previous larger time component. Omit all parameters to default to the current system time.

SystemTimeToSeconds

Takes a System.DateTime object and returns the total number of seconds represented by the object. Omit all parameters to default to the current system time.

TimeToDegrees

Converts a floating point time in seconds to an angle in degrees, typically for use by clocks and meters in both 2d and 3d. Each optional boolean represents the resolution at which to represent the time value. This determines how "smooth" the clock hand will move. By default the method will return an angle corresponding to the current amount of seconds in the current minute at millisecond resolution.

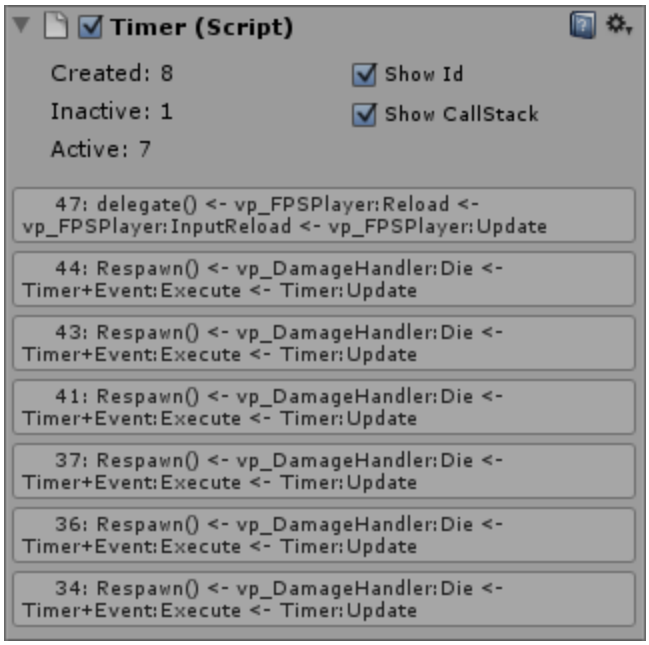
SystemTimeToDegrees

Converts a System.DateTime object into a Vector3 representing three degree angles: **x** representing the hour-hand, **y** representing the minute-hand, and **z** representing the second-hand of a classic analog clock, respectively. Omit the **time** argument to default to the current system time.

```
// --- EXAMPLE 1: use the current system time
Vector3 angle = vp_TimeUtility.SystemTimeToDegrees();

// --- EXAMPLE 2: use a made-up time
Vector3 angle = vp_TimeUtility.SystemTimeToDegrees
                    (new System.DateTime(1, 1, 1, 16, 37, 50, 0));
```

Debug Mode



Compiling visionTimer with the **DEBUG** define will allow you to display debug info on all created timers in the Unity Inspector. This is very useful when debugging complex gameplay. Open the file 'vp_Timer.cs' and at the top of the file uncomment this line:

```
// --- EXAMPLE: run vp_Timer in debug mode
#define DEBUG     // uncomment to display timers in the hierarchy
```

Press play in Unity and you should notice a new gameobject **"Timers"** in the Hierarchy upon the first timer event initiated. Select this object and a **"Timer"** component will become available in the Inspector. Every active timer will be represented in this view as a box with the name of the method being called, its arguments, and optionally the id and callstack info of the function call. When a timer event triggers, it will return to the object pool and disappear from view.

- When done debugging, remember to comment out the **DEBUG** define again! visionTimer feeds a lot more junk to the garbage collector in this mode.

Debug info

Created

Shows the total amount of timer event objects created during this session. This will be the same as the maximum number of timer events ever having been active simultaneously (see "About the object pool" for more info on this).

Inactive

This number represents the amount of timer events that have been recycled and may be used for subsequent scheduling.

Active

This number represents the amount of currently active timers.

Show Id

Displays the **Id** of the *call* to vp_Timer.In or vp_Timer.Start that (re)activated this timer event. This is an ever growing number, i.e. each **Id** of such call is unique.

Show Callstack

Displays a stack trace (a list of the function calls that led up to the timer event being initiated). This is very useful for debugging, especially if you are using lots of delegates and have no method name to go by.

> ***TIP:*** *If you really need to name delegates for debug purposes, schedule them using a string with a name of your choice as the argument. The nametag will be visible in the list between the parentheses of the delegate.*

About the object pool

Some gameplay types have an awful lot of timers firing in rapid succession. Even if memory allocation is minimal, over time this can lead to fair amounts of data being garbage collected which is something you really want to avoid in Unity since it will lead to framerate hiccups over time (especially on mobile platforms).

vp_Timer uses an object pool under-the-hood in order to feed the garbage collector with an absolute minimum amount of data. This means that once a timer event has executed, it is recycled and put in the object pool, to be used by future calls to vp_Timer.In or Start. It also means that the number of timer event objects ever created will never exceed the maximum number of timer events ever having been active simultaneously. In short, it leads to super limited object creation.

- Even if the timer logic itself is gentle on the garbage collector, the objects you create inside your methods and delegates still get fed to the garbage collector. To avoid stuttering framerate in Unity, it's a good idea to be moderate when using the **new** keyword and creating objects in general.

- If you for some reason want to kill all the under-the-hood objects created by vp_Timer, use **vp_Timer.DestroyAll**. As long as you remember to also dereference any created Timer.Handles too, this should clear all currently active timers along with the ones in the object pool and release memory to the garbage collector.

Support and additional information

Make sure to check out Ultimate FPS Camera for many examples of vp_Timer being used in an advanced project.

If you have any further questions or just want to show off your work, don't hesitate to participate in the Unity Community forum discussion:
http://forum.unity3d.com/threads/157013-VisionTimer-RELEASED

... or fire off an email to: info@visionpunk.com

Happy game programming =)

/Cal