GAMECAVE EFFECTS ENGINE 3.X

DOCUMENTATION/MANUAL

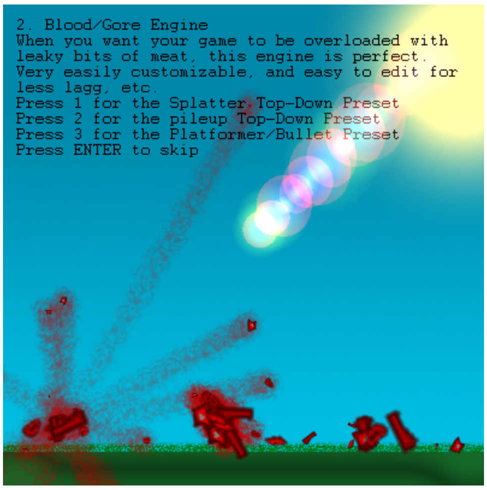
LAST UPDATED: 4TH MARCH 2008 - VERSION 3.1

TABLE OF CONTENTS (CLICK TO VIEW CHAPTER)

INTRODUCTORY

Many elements in game-design make up the overall quality and performance in video games and other software designed to entertain and/or amuse the player. Arguably, one of the most important game-design elements lies in the game's *graphic /appearance* quality*.* As many games are being released both commercially and freely over the web, graphic quality plays an important part in making your game stand out from the crowd. Of course, this can be an extremely daunting task, even with professional graphic-design skills. Luckily, there's a less complex method of improving your game's visual quality.

The *GameCave Effects Engine* has been running since Early 2005, and has since been downloaded over 15,000 times, helping programmers encase their projects with a rich polish of visual effects with little programming knowledge, or without the best graphic-design skills. The engine provides over 20 mini-engines, all devoted to providing a great visual effect. Some are based on realism, others with a more cartoonist approach. As the *Effects Engine* is completely editable, all effects can be manipulated and re-designed to fit any type of game; performance, quality, and style wise.



```
2. Blood/Gore Engine
When you want your game to be overloaded with
leaky bits of meat, this engine is perfect.
Very easily customizable, and easy to edit for
less lagg, etc.
Press 1 for the Splatter Top-Down Preset
Press 2 for the pileup Top-Down Preset
Press 3 for the Platformer/Bullet Preset
Press ENTER to skip
```

Though use of the *effects engine* requires some credit for their work (see *License*), it is completely free to use, and all coding is completely commented, allowing you to easily understand the engines you wish to use. And, as of version 3.0, the original particle-designer *PartArt 2.0* is available, integrated into the Effects Engine. *PartArt* allows you to use a graphical-user-interface (GUI) to design your own particles. The program generates the particle-coding as you work on it, so there is no coding knowledge needed to let your imagination journey across the potential of particle effects in *GameMaker*.

Also included with the *GameCave Effects Engine* are various tutorials for functions and systems used in *GML* that help your game's visual appearance. Such tutorials include *Particle Effects*, *Surfaces*, and *Alpha Mapping.*

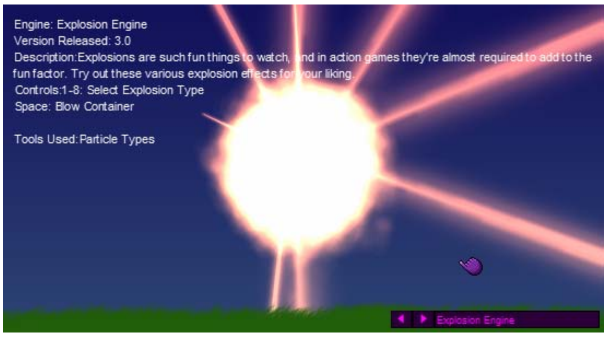
EFFECTS INFORMATION

There are many effects included with the *GameCave Effects Engine* package. Each effect is heavily commented, and implementation-friendly – so that it's relatively easy to install into your own game. Of course, every game is different and so problems can always arise in installing these effects. If this is the case, you should read this chapter to further understand how the effect works, in order to know what may be causing conflict between your game and the effect you wish to use. Of course, if this doesn't work, use the contact details to contact us and we will help you in any way possible.

All the objects and resources explained in each effect are available in the "GCE_(effectname)" group of the resource folder(s).

EXPLOSION

The *explosion* effect is the longest lasting visual effect in the GC Effects Engine – it has been persistent in the Effects Engine since v1.0. However, since v3.0, the explosions have been revamped and optimized. Explosions are extensive processes, and if you over-use them, it can slow down your game.



Engine: Explosion Engine
Version Released: 3.0
Description:Explosions are such fun things to watch, and in action games they're almost required to add to the fun factor. Try out these various explosion effects for your liking.
Controls:1-8: Select Explosion Type
Space: Blow Container

Tools Used:Particle Types

To control the explosion effect in the GCE Browser, press any number from 1-8 to select one of the 8 explosions, and then press SPACE to see the explosion. To see another explosion, just press another number.

These controls are coded with the **objExplosion_control** object. The **index** variable in the object determines which explosion to bring up for exploding. If the value is 0, no explosion is ready. Otherwise, the value is equal to which explosion number (1-8). The step event simply checks for the key-presses and switches the index value (as well as switches the visibility of explosion barrels – this does not affect whether the actual effect is visible or not). It also returns the index to 0 if space is pressed.

The other objects in the explosion group are responsible for 1 explosion effect each. The number at the end of the object indexes show which explosion number they're responsible for. In each create event for these explosions are the defining of the particle effects for the explosion.

Key functions in the create event include
part_system_depth – change the second argument to choose the depth of the explosion effect.

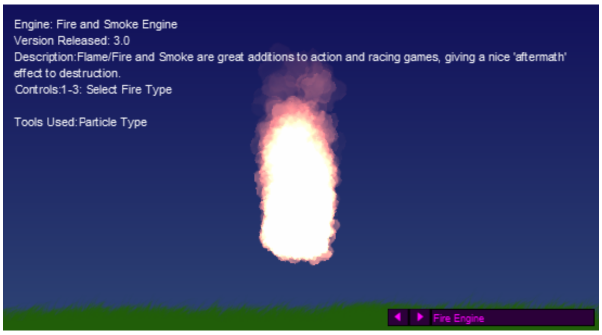
part_emitter_region – Change the third, fourth, fifth, and sixth argument to the **xmin, xmax, ymin, ymax** (in order) of where the explosions should be burst. So, for instance, if you want the explosion effect to burst at the X and Y position of the object defining the particles, you would change the arguments to (in order) x,x,y,y.

If you wish to edit any other functions in the create event, consult the GM Manual or the GCE Particles Tutorial (included with this package) for more information.

The space event in each object first checks whether the **index** of the control object is equal to the explosion number the object is responsible for, and if it is, the visual effect is put into action, and the explosion barrel is then made invisible. The key functions in this event include the **part_emitter_burst** functions. These burst a number of particles for the effect. Each repetition of the function is for each type of particle (as various explosion effects have various types of particles – such as smoke and fire).

In the case of explosion **7** and **8,** there is also the setting of an **alarm** when the space bar is pressed. This is for the beams of light that shoot out of the explosions – looking in the **alarm0** event, a **‘times’** variable is decreased when the alarm goes off, and then the alarm is reset. The light-beam is then burst each time the alarm goes off.

FIRE



Engine: Fire and Smoke Engine
Version Released: 3.0
Description:Flame/Fire and Smoke are great additions to action and racing games, giving a nice 'aftermath' effect to destruction.
Controls:1-3: Select Fire Type

Tools Used:Particle Type

The *fire* effect is similar both structurally and visually to the *explosion* effect. It is built with particle effects and is also controlled by an index variable in **objFire_control**. The **index** variable is equal to 0 if no fire effect is being burst (pressing space will do this), else it is equal to the fire effect to burst. The step event of **objFire_control** sets the index variable according to number keys being pressed. Then, the **objFire_(number)** objects are responsible for the fire effects (according to the number on the end of their object index – i.e objFire_1 is responsible for the first fire effect).

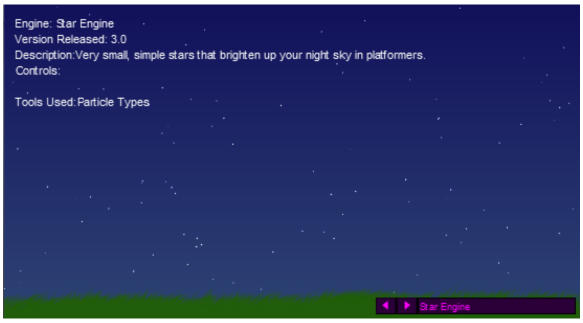
Key functions in the create event include
part_emitter_region – Change the third, fourth, fifth, and sixth argument to the **xmin, xmax, ymin, ymax** (in order) of where the fire should be present. So, for instance, if you want the fire effect to be at the X and Y position of the object defining the particles, you would change the arguments to (in order) x,x,y,y. **Note:** Seeing as the function is only in the create event, the region arguments are only set *once*. If your arguments include “x” and “y”, acknowledge it does not update the x and y position unless you place the function in the *step* event.

If you wish to edit any other functions in the create event, consult the GM Manual or the GCE Particles Tutorial (included with this package) for more information.

In the step event of the **objFire_(number)** objects, the object checks whether **objFire_control’s** index variable is set to the fire effect this object is responsible for. If so, it bursts a batch of fire particles. The **part_type_orientation** function is continually set to provide a random angle-increment for the particles.

STAR

The first star effect, though very subtle and static, can be a simple and nice addition to any night-sky in a platformer. The stars throb very slightly to depict twinkling. The stars are made with particles that last 1,000,000 steps (so that they don't go away).

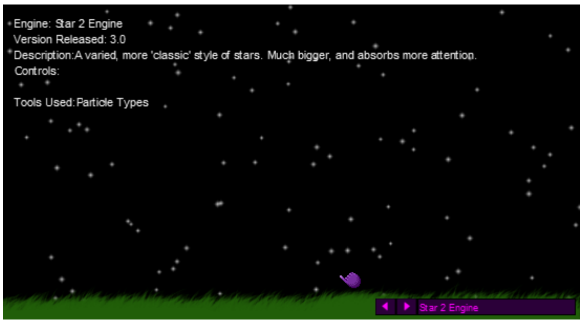

Engine: Star Engine
Version Released: 3.0
Description:Very small, simple stars that brighten up your night sky in platformers.
Controls:

Tools Used:Particle Types

Stars are relatively easy to implement without problems as only one object is used to design, display, and remove the stars. That object is **objStars**. The create event both declares the particles, and bursts them. The room-end event then destroys the particles from the game.

STAR 2

The second star effect is a little more vibrant. It shows throbbing white stars on a black background – This star effect shows more of a *classic* style.



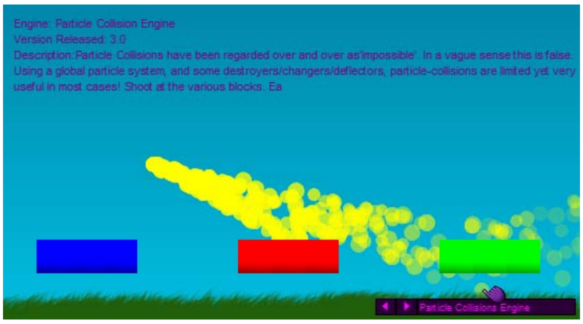
Engine: Star 2 Engine
Version Released: 3.0
Description:A varied, more 'classic' style of stars. Much bigger, and absorbs more attention.
Controls:

Tools Used:Particle Types

This star effect has an almost exact structure as the first star effect. The stars are made with particles that last 1,000,000 steps (so that they don't go away).

Stars are relatively easy to implement without problems as only one object is used to design, display, and remove the stars. That object is **objStar2**. The create event both declares the particles, and bursts them. The room-end event then destroys the particles from the game.

PARTICLE COLLISIONS



Engine: Particle Collision Engine
Version Released: 3.0
Description: Particle Collisions have been regarded over and over as'impossible'. In a vague sense this is false. Using a global particle system, and some destroyers/changers/deflectors, particle-collisions are limited yet very useful in most cases! Shoot at the various blocks. Ea

Particle Collisions are considered impossible (at least with GameMaker's current structure) – and in a large respect, this is true. However, with GM's built-in deflector, destroyer, and changer functions, and the use of a "global" particle system, it's rather easy to simulate collisions with particles. This effect shows how simple particles can collide in 3 different ways (with 3 different coloured boxes).

There are 4 objects that make up the particle-collision effect. One object controls the particles, systems, and emitters (**objParticle_collisions**), and the other 3 are the 3 types of blocks, which control their own "particle-collision mask" (**objPC_wall_deflector, objPC_wall_destroyer,** and **objPC_wall_destroyer**).

objParticle_collisions – In the create event, we create a particle system, and within that system we create 3 emitters: one for the left side (left-click), one for the middle (middle-click), and one for the right side (right-click). When implementing into your game, this is irrelevant as the controller object is mainly for displaying how the collisions work. We then have 2 particles. One is the normal particle type, shot out when pressing one of the 3 mouse buttons. The other is the particle-type that is replaced with the other particles, when they collide with **objPC_wall_changer** (the blue block). Finally, we have 3 **with** statements. This executes the user-events of the 3 block-types, telling them to create their particle-collision masks (more on that event later).

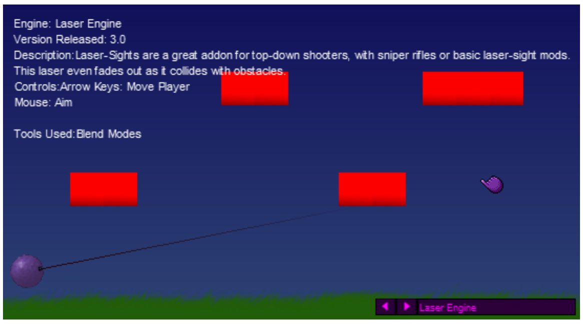
The room-end and destroy events are used simply to destroy the system (including all particle-collision masks and emitters) when the effect is finished. We then just have the three "global mouse button" events. Each one bursts the main particle type with one of the 3 emitters, also changing the particles direction function (**part_type_direction**) to fit with the new mouse position.

The other 3 objects, **objPC_wall_changer, objPC_wall_deflector,** and **objPC_wall_destroyer,** all have similar structures. In their create event, they set their image_blend variable to a particular colour (blue, red, or green) – this is just part of the example. The *other* event is a user-event (0). This event, as said previously, is executed via **objParticle_collisions** create event, with a ***with*** statement. This is to ensure that the particle-collision masks aren't created before the particle system is, else an error would return not being able to find the system to create the collision masks.

In these user-events, a deflector, changer, or destroyer is created (according to the object). Then, a region that covers the block's width/height is applied to that deflector/changer/destroyer (creating the collision-region), and then more settings are applied, like friction/force for deflectors, and particle-types to change to for changers. For more information on those functions, consult the GM Manual or the GCE Particles Tutorial included with the package.

LASER

The laser effect, or “laser-sight effect”, is nice for top-down-shooters. It uses one simple group of functions to draw a fast, accurate, and visually-attractive laser-pointer. The laser has a maximum range of 800 pixels, but can be shortened if it collides with a wall.



Engine: Laser Engine
Version Released: 3.0
Description: Laser-Sights are a great addon for top-down shooters, with sniper rifles or basic laser-sight mods. This laser even fades out as it collides with obstacles.
Controls: Arrow Keys: Move Player
Mouse: Aim

Tools Used: Blend Modes

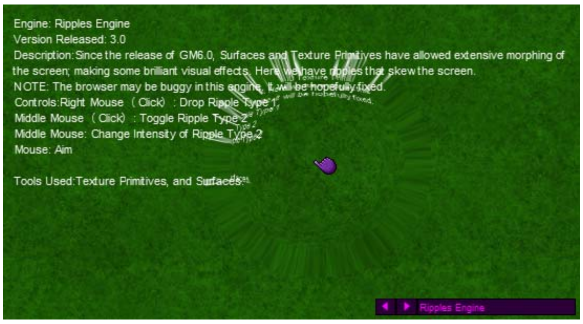
There are 3 objects to exemplify the laser-effect. **objLaser_ball** (the ball that you control and point the laser with), **objLaser_messenger** (an object that looks for the position that the laser must end), and **objLaser_wall** (a simple wall for the laser to collide with). All the coding, however, is within the **objLaser_ball** object. Firstly, the ball is put in the middle of the room (see create event). Then, the arrow-key events can move the ball around. Finally, the draw event – this draws the ball and calculates where to draw the laser (then, of course draws it).

The **with** statement creates a new messenger object, which uses the **move_contact_solid** object to find any solid object (**objLaser_wall,** for example) within 800 pixels away from **objLaser_ball**, in the direction of the mouse. It then sends it’s result to the **laser_x** and **laser_y** variables in **objLaser_ball**. The messenger then destroys itself, and the ball is left to draw the red/black line (in subtractive blend modes, to make the line fade out) from it’s x/y position, to the x/y position in variables **laser_x** and **laser_y**, left by the messenger.

This is all that’s required to use the laser-line. Change “`point_direction(x,y,mouse_x,mouse_y)`” to the direction which you want the laser to point at.

RIPPLES

Ripples use surfaces and primitive textures to stretch and morph the screen in certain areas, to provide various effects. There are 2 types. The first type bulges (or “sucks in”) part of the screen with a given intensity. The second type makes a “drop” effect, which gets larger and fades out.



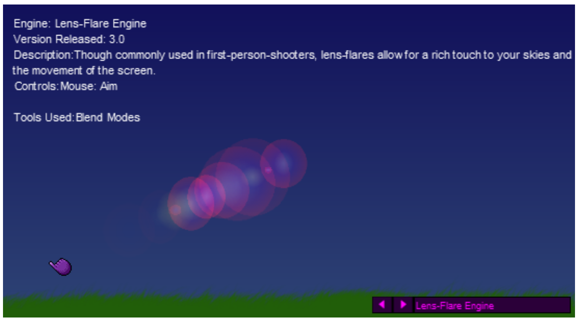
Engine: Ripples Engine
Version Released: 3.0
Description: Since the release of GM6.0, Surfaces and Texture Primitives have allowed extensive morphing of the screen; making some brilliant visual effects. Here we have ripples that skew the screen.
NOTE: The browser may be buggy in this engine. It will be hopefully fixed.
Controls: Right Mouse (Click) : Drop Ripple Type 1
Middle Mouse (Click) : Toggle Ripple Type 2
Middle Mouse: Change Intensity of Ripple Type 2
Mouse: Aim

Tools Used: Texture Primitives, and Surfaces.

Surfaces and texture-primitives can be a very sensitive feature of GameMaker. It can be buggy, really slow, or simply not working by the slightest of mistakes. Different video card chipsets, if too old, can cause weird problems with surfaces, and so you should use this effect with care.

In the ripples engine, there are 3 objects. **objRipple_control**, which controls the main surface and which ripple object to use – then, the two ripple-type objects, **objRipple_1** and **objRipple_2**. In **objRipple_control**’s create event, there is the *surface_create* function. It is recommended that, if you have a view, you set the second and third arguments to the width and height of the view. The size of a surface really affects the memory usage of your game. The global mouse buttons switch between which ripple-type object is created – while limiting one instance of that object in the room at once. The comments in those events cover the events.

LENS FLARE


Engine: Lens-Flare Engine
Version Released: 3.0
Description:Though commonly used in first-person-shooters, lens-flares allow for a rich touch to your skies and the movement of the screen.
Controls:Mouse: Aim

Tools Used:Blend Modes

The lens flare is a very simple effect that shows a first-person-shooter style lens-flare. All it uses is a group of blended sprites.

There are 2 objects that make up the lens flare. They're both responsible for two sprites of the lens-flare, but work similar other than that. The 2 objects are **objLens_1** and **objLens_2**. In their create events, it's important you replace these two lines:

```
x = room_width/2;
y = room_height/2;
```

With the x/y positions you wish to set the lens-flares to (not where they point towards, but where they start from – for instance, a sun). If you want them to be where you place them, just remove the two lines.

Next, with a bit of GML knowledge, you can edit some of the variables in the create event to make different looks of the lens-flare (for instance, adding more flares or changing their alpha values).

You then have the *draw* event. This draws each lens-flare. Be sure to replace all instances of "mouse_x" with the X position you wish to point the lens-flare towards, and all instances of "mouse_y" with the Y position.

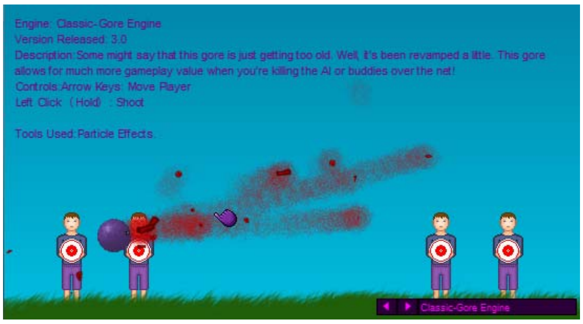
EARTHQUAKE

The earthquake effect shakes the view's X and Y positions at certain intensity, keeping in mind the original x and y position so that it can safely return to that position after the earthquake is finished. This effect is great for explosion shakes, and really adds to the depth of your game.

The whole earthquake effect is controlled by one object, **objEarthquake_control**. In the create event, you have a few variables to determine how the earthquake will work. The comments after each variable will explain what the variable does.

Nothing should be changed in the step event, so we can switch straight to the up/down key events. They simply change the **shake** variable, increasing/decreasing the intensity of the earthquake. The *space* event switches the **doit** variable between 1 and 0 (switching the earthquake on/off).

CLASSIC GORE



Engine: Classic-Gore Engine
Version Released: 3.0
Description:Some might say that this gore is just getting too old. Well, it's been revamped a little. This gore allows for much more gameplay value when you're killing the AI or buddies over the net!
Controls:Arrow Keys: Move Player
Left Click (Hold) : Shoot

Tools Used:Particle Effects.

"Classic Gore" has been with the Effects Engine since the very early versions. It has provided nice gore effects for many games. Since GCE 3.X, the gore has been updated and improved slightly. There are **11** objects in the classic-gore engine, all of which will need some explanation. There are 4 gore-effects, with 2 types and 2 sub-types for each type. The first type is for top-down games – the blood sprays without gravity, and slides before stopping. The second type is for platformers – the blood sprays, and falls to the ground. The two sub-types determine whether the blood flies relative to the bullets direction, or whether it shoots out randomly. The objects consist of:

objClassicGore_controller – Controls the little ball you can control/shoot with in the example.
objDummy1 – Emits the first type (with sub-type A) of gore.
objDummy1b – Emits the first type (with sub-type B) of gore.
objDummy2 – Emits the second type (with sub-type A) of gore.
objDummy2b – Emits the second type (with sub-type B) of gore.
objClassicGore_bullet – A bullet created by pressing the left mouse button. It creates different types of gore (and sets it's direction if appropriate) according to collisions detected for each dummy object.
objClassicGore1_stain – A "stain" (trail of blood) that is left behind by the pieces of gore.
objClassicGore1 – A piece of meat/gore for the first type (with sub-type A) of gore.
objClassicGore1b – A piece of meat/gore for the first type (with sub-type B) of gore.
objClassicGore2 – A piece of meat/gore for the second type (with sub-type A) of gore.
objClassicGore2b – A piece of meat/gore for the second type (with sub-type B) of gore.

As **objClassicGore_controller** is documented enough in its code, and does not play an important part in implementing into your own games, it's not documented in this chapter. This also applies to each **dummy** object, as they do not have any code to execute. You simply need to acknowledge that **objClassicGore_bullet** creates various pieces of gore, according to which **dummy** object it finds in its path.

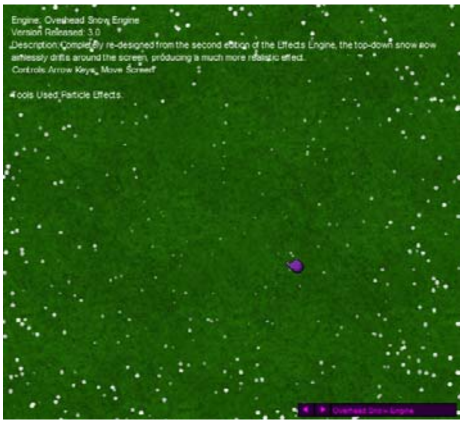
objClassicGore_bullet, however, requires some explanation. The create event simply sets the image_angle to face where it's going, and sets a speed. The draw event draws a line between its' current X and Y position, and it's X and Y position from the previous step (in which case, it's a line 15 pixels long, as the bullets speed is 15 pixels a step).

We then have 4 **if** statements, each being responsible for a collision line between the bullets current X and Y position, and it's X and Y position from the previous step (the same as the line that's being drawn) – each if statement being for 1 of the 4 dummy objects. According to that dummy object, it creates the according gore object (in a **repeat**(3) statement – thus creating 3 gore objects). The bullet then destroys itself. In the case of colliding with either **objDummy1b** or **objDummy2b**, a **with** statement is used with the newly created gore piece, setting its direction to the **other** direction (other being the bullet) with a random factor

Next we have **objClassicGore1_stain**. The create event sets some parameters for the motion and look of the stain. This includes the angle, a random subimage, 0.2 alpha, and an alarm to set the **fade** variable to true. The **fade** variable, if set to true, will tell the stain to start fading out its alpha (concluding in the instance's destruction). The step event fades out the alpha (if **fade** is true), and destroys if the alpha is 0 or less. Finally, the outside room event destroys the stain too.

All the gore objects are also well commented inside their events – and do not need any further documentation. They simply move according to some random parameters set in their create event, create blood-stains in the alarm 1 event (which continually resets itself) if it's going fast enough, and fades out after **fade** is set to true. Any collision events with objGrass is for gravity-affected gore-pieces (platformer blood – type 2), which stops the gore from moving further.

OVERHEAD SNOW



Engine: Overhead Snow Engine
Version Released: 3.0
Description: Completely re-designed from the second edition of the Effects Engine, the top-down snow now aimlessly drifts around the screen, producing a much more realistic effect.
Controls: Arrow Keys, Move Screen

Tools Used: Particle Effects.

The overhead-snow effect depicts realistic snow for top-down shooters. It uses basic particles, and 4 emitters (1 for each side of the screen) to burst the snow. It works rather simply, and is controlled all in one object, **objOverheadsnow**.

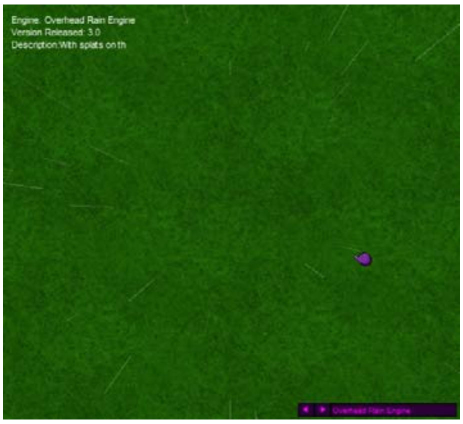
The create event first creates a particle system, and 4 emitters – one for each side of the view. Make sure your room has its view enabled. If this is a problem, use the GM Manual (or the GCE Particles Tutorial) to learn the **part_emitter_region** function, to edit the function so that it considers room edges instead of view edges. After declaring each emitter, and applying the regions (specifying where to burst the snow), the snow particle is created and defined.

The step event then redefines the direction (to random parameters) for the snow 4 times (one for each side), to provide a bit more randomization in the snow. It then bursts 1 particle of snow for each side/emitter (4 particles in total). We then check for key-presses, and move the view accordingly. Finally, the emitter regions are re-applied, to keep synchronized with the view’s x/y position.

Finally, the room event destroys the snow system to free some memory.

OVERHEAD RAIN

Though the overhead rain would seem similar to the snow effect, it is in fact quite different. The overhead rain effect has 3 objects that make up the final result. **objOverheadrain**, which moves the view and creates the rain drops, **objDrop**, a single drop of rain moving towards the centre of the screen, and **objDrop_splat**, the little splosh animation when the drop hits the ground. The rain effect doesn't use any particles, just primitive drawing and instance creation.



The step event of **objOverheadrain** first checks for keyboard-arrow presses, and moves the view accordingly (part of the example). Then, it creates a rain-drop (**objDrop**) on 4 sides of the view (with random positions – for instance, a rain-drop being created at the top could be created anywhere from the top-left corner of the view to the top-right). This requires you have view enabled in your room. If, for any reason, you can't have views enabled (and view 0), Replace all "view_xview[0]" and "view_yview[0]" lines in the step event with '0' (without quotes).

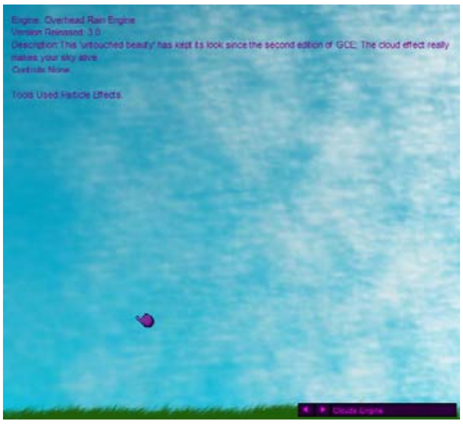
objDrop sets a lot of parameters in its create event to determine it's look, and it's destination in the room. The **distance** variable is used to determine a random distance (from it's start position – in pixels) to where it splashes/hits the ground. **Image_xscale** is set in the create event, as it decreases as the drop falls (to give the illusion of the drop falling further away from the screen). **Destx** and **desty** is used to determine the direction for the drop. Of course, the drop won't die when it reaches these coordinates, but will fall short according to the **distance** variable.

The destroy event simply creates **objDrop_splat** at its position, so we can see the drop splash on the ground.

The step event is used to reset the image_xscale, so that the drop can get smaller as it reaches its death-spot. It also destroys the drop if it has reached its destination.

objDrop_splat works very simply. The create event picks a random alpha value (for a bit of variation between splashes), and the step event fades out the alpha value (eventually destroying the object).

CLOUDS



Engine: Overhead Rain Engine
Version Released: 3.0
Description: This 'untouched beauty' has kept its look since the second edition of GCE. The cloud effect really makes your sky alive
Controls: None

Tools Used: Particle Effects.

The clouds effect is a classic favourite, being implemented in the very early versions of the effects engine. It simply show's a platformer/side-scroller style cloud-scrolling effect, using particles. The clouds effect can sometimes be slow, but there are some simple methods of sacrificing a bit of appearance value for a performance boost, that will be explained below.

One object controls the cloud effect – **objClouds**. The create event defines all particles and their systems/emitters, and then starts continually streaming the clouds. There are 2 particle systems, **cloud_infront** and **cloud_behind**. This is to give the clouds some depth, by creating smaller clouds that move slower, behind the larger clouds. It will give a parallax effect. Then, two cloud-types are created – **cloud0** and **cloud1**. **cloud0** is for the **cloud_infront** system, and **cloud1** is for the **cloud_behind** system. To improve performance, use the **part_type_size** functions, by making the second and third arguments smaller – such as:

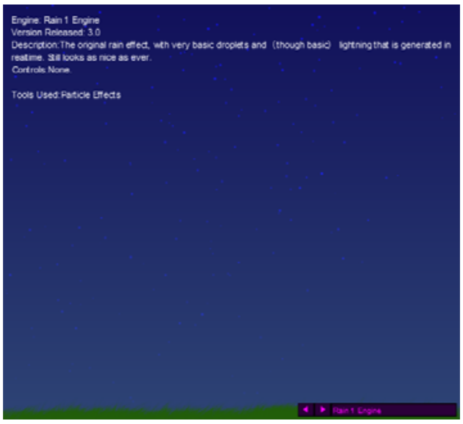
```
part_type_size(cloud0,1,2,0,0.02)
```

into

```
part_type_size(cloud0,0.7,1.2,0,0.02)
```

Finally, the emitters are created (1 for each system), and each emitter streams the respected particle. To make it seem like the clouds have “always been there” when the room starts (instead of having the clouds slide on the screen, with a blank screen at the start), the **repeat** statement has been used to update the particle’s position/look 400 times/steps (to ‘fast forward’ the process).

RAIN 1



Engine: Rain 1 Engine
Version Released: 3.0
Description:The original rain effect, with very basic droplets and (though basic) lightning that is generated in realtime. Still looks as nice as ever.
Controls: None.

Tools Used:Particle Effects

In the Effects Engine, there are 3 types of rain. This, first effect is a simple, more cartoonish approach to a rain *and* lightning effect. The rain uses particles, where as the (realtime-generated) lightning uses basic drawing primitives to generated a unique shape every time lightning strikes.

There are 3 objects that make this effect. **objRain1**, which controls all the rain particles and randomly creates lightning, **objThunder_fork_basic**, which makes one instance/strike of lightning, and **objThunder_sheet**, which makes the screen flash when lightning strikes.

The create event of **objRain1** defines all the rain particle parameters, and starts streaming them (5 per step). You can change the intensity of the rain by changing the ‘5’ to a higher or lower number (in the **part_emitter_stream** function). If your room is very large, and uses a view, it may be wise to change the **part_emitter_region** arguments to ‘**view_xview[0]**’ (instead of 0), and ‘**view_xview[0]+view_wview[0]+200**’ (instead of room_width+200). Finally, the rain-drops are automatically moved 50 steps before they become visible, so that the rain is already streaming normally when the room starts.

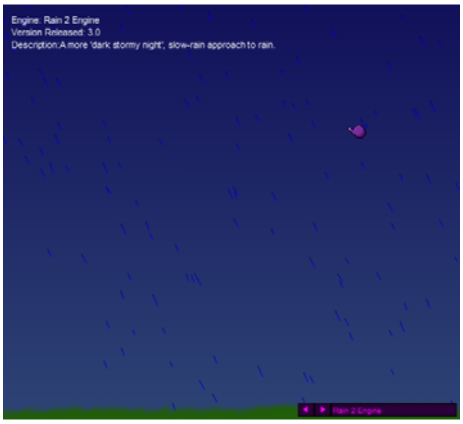
The destroy event (and room end event) simply destroys the particle-system for the rain, to free memory.

The step event randomizes the chance of lightning being created (a chance of 1 in every 80 steps is given), and creates the other two objects (**objThunder_fork_basic**, and **objThunder_sheet**) accordingly. If you are using a view, you may want to change “random(room_width)” with “**view_xview[0]+random(view_wview[0])**”. If you want to use the complex lightning-style (see RAIN 3), simply replace ‘**objThunder_fork_basic**’ with ‘**objThunder_fork_complex**’.

objThunder_sheet draws a white rectangle all over the screen, which fades in and out quickly. The create event makes the alpha set to 0 (invisible), and sets an alarm that will destroy the sheet when it’s finished flashing. The step event increases alpha until It has reached full – alpha (fully visible). It then switches the **fadein** variable, which makes the alpha start fading out again (back to 0). Finally, the draw event draws the sprite (1x1 sprite), stretching it across the whole room.

objThunder_fork_basic isn’t as basic as it seems. Though, anything you may want to change in lightning can be done through variable values that are placed in the create event. They are explained enough by their comments. The draw event draws the lightning according to the variables declared in the create event.

RAIN 2



Engine: Rain 2 Engine
Version Released: 3.0
Description:A more 'dark stormy night', slow-rain approach to rain.

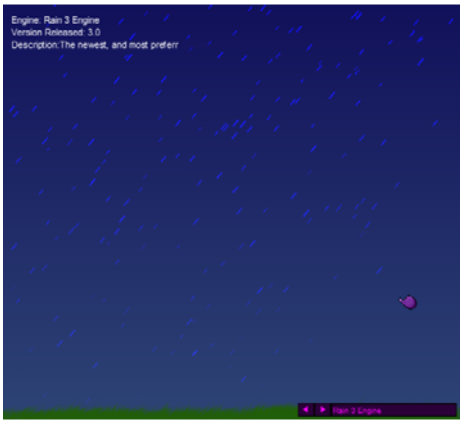
The second rain effect has a slightly similar look to the first, except this depicts a more realistic look. This effect does *not* use particles, so there is 1 object instance for every drop. This makes the game more compatible, and allows rain drops to have collisions (further than what the **particle collisions** engine is capable of) as well as execute normal object events, but may show more of a performance hit later on.

This time, there are just 2 objects that make up this effect. **objRain2_control** creates all rain drops appropriately, and **objRain2_drop** is an individual drop that is created by **objRain2_control**.

In **objRain2_control**'s step event, we simply create (3) drops at a random spot across the room. If you are using a view, it is recommended you change the first argument of instance_create to "**view_xview[0]-100+random(view_wview[0]+100)**".

objRain_drop is also pretty basic. The create event specifies some random speeds and directions of the rain drop, to give some variety and depth. The draw event then draws a line from the rain-drops current position, to the rain-drops previous position. This means, the faster the rain-drop is going, the shorter the line.

RAIN 3



Engine: Rain 3 Engine
Version Released: 3.0
Description:The newest, and most preferr

The final rain effect is the most preferred rain effect. It uses both particle effects, and texture primitives to create a complex alternative to lightning, which uses advanced mathematical calculation to generate one, realistic and unique look of lightning every time.

This rain effect has 3 objects to make up the effect. Firstly, we have **objRain3,** which defines and creates the rain particles, and then we have **objThunder_sheet** again, along with the other lightning-effect-type, **objThunder_fork_complex**.

The create event of **objRain3** defines all the rain particle parameters, and starts streaming them (5 per step). You can change the intensity of the rain by changing the '5' to a higher or lower number (in the **part_emitter_stream** function). If your room is very large, and uses a view, it may be wise to change the **part_emitter_region** arguments to '**view_xview[0]**' (instead of 0), and '**view_xview[0]+view_wview[0]+200**' (instead of room_width+200). Finally, the rain-drops are automatically moved 50 steps before they become visible, so that the rain is already streaming normally when the room starts.

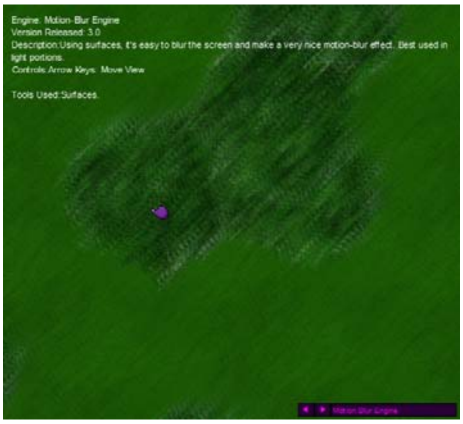
The destroy event (and room end event) simply destroys the particle-system for the rain, to free memory.

The step event randomizes the chance of lightning being created (a chance of 1 in every 80 steps is given), and creates the other two objects (**objThunder_fork_complex**, and **objThunder_sheet**) accordingly. If you are using a view, you may want to change "random(room_width)" with "**view_xview[0]+random(view_wview[0])**". If you want to use the basic lightning-style (see RAIN 1), simply replace '**objThunder_fork_complex**' with '**objThunder_fork_basic**'.

objThunder_sheet is explained in the "Rain 1" chapter.

Finally, we have **objThunder_fork_complex**. The name speaks for itself, the coding is rather complicated. However, to manipulate the lightning, most of it can be done with a few variable changes, in the create event. These variables are explained through their comments. The draw event simply draws the randomly generated lightning.

MOTION BLUR



Engine: Motion-Blur Engine
Version Released: 3.0
Description:Using surfaces, it's easy to blur the screen and make a very nice motion-blur effect. Best used in light portions.
Controls:Arrow Keys: Move View

Tools Used:Surfaces.

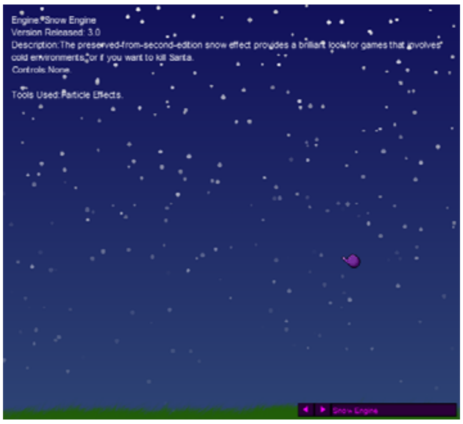
Motion blur is a great effect that can be implemented in many ways, in many games. Examples would include high-speed racing, or being hit by bullets in a shooter game. Though this motion-blur effect can be considered rather demanding in terms of performance, most average computers these days should be able to run the effect without any problems. However, a video card with *at least* 64MB of memory is required.

To use the effect, you will usually need rather advanced skills with GML, to keep the visual effect running properly. There is just 1 object that makes the motion blur effect. It is called **objMotionBlur**

The create event creates *2 surfaces*, with the width and height of the region's size. Then, some parameters are set, such as **blur_amount** (the "alpha" of the screen – the less the alpha, the more you can see the blur. 0 will freeze the screen, 1 will provide no blur). We then use **set_automatic_draw** to 0, meaning we need to manually choose when to redraw/refresh the screen.

The end-step event first stops the view from moving off the screen, and applies speed/friction on its movement. Then, the 2 surfaces take a "snapshot" of the screen, applies some unique blend modes to it, and draws the surface with the specified blur amount. **This should not be tampered with**. The 4 keyboard events move the view-speed appropriately, and the room end event sets **automatic draw** back to 1, and frees (removes from the memory) the 2 surfaces, as we have finished using the motion-blur.

SNOW



Engine: Snow Engine
Version Released: 3.0
Description: The preserved-from-second-edition snow effect provides a brilliant look for games that involves cold environments, or if you want to kill Santa.
Controls: None.

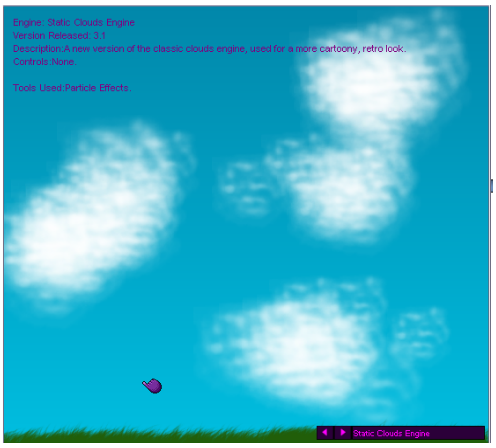
Tools Used: Particle Effects.

The final effect in the effects engine is this ‘veteran’ snow effect – surviving untouched in the Effects Engine since the early versions. The snow effect uses pure particle effects, and is all controlled by one object: **objSnow**.

The create event of **objSnow** defines all particles, their systems, and the emitter. It then starts streaming 2 particles every step, fast-forwarding the particles movement 100 steps so that the snow is already half-way down the screen when the game starts. If you are using a view, it is recommended that you change “-500” in the **part_emitter_region** function to “view_xview[0]-500”, and “room_width+500” to “view_xview[0]+view_wview[0]+500”.

The destroy and room-end events simply destroy sthe particle system.

STATIC CLOUDS


Engine: Static Clouds Engine
Version Released: 3.1
Description:A new version of the classic clouds engine, used for a more cartoony, retro look.
Controls:None.

Tools Used:Particle Effects.

This is an alternative to the classic cloud effect. This effect has lots of small clouds that do not move, as opposed to one large smog of cloud drifting across the sky. The whole effect is controlled by one object: **objStaticCloud**.

In the create event, you can change the number of clouds and the intensity of each cloud with the respective variables.

Tutorial Information

Included with the GCE 3.X package is a series of tutorials to help you design your own visual effects in *Game Maker*. Such tutorials include creating particle effects, using alpha masks, blend modes, and more.

To view these tutorials, simply open the appropriate PDF file included with the GCE Package. You must have Adobe Acrobat Reader, available for free from www.adobe.com.

GCE LICENSE

GameCave Effects Engine Version 3.X – Written by Rhys Andrews

All resources, including sprites, graphics, scripts, objects, and anything included in the GameCave Effects Engine Package is free of use, with the exception of acknowledgement and credit permission from the respected owner(s). Please contact us via **email** or **Gm-Community PM**, available in the *Contact Details* chapter. Credits must include the text *“Effects by GameCave Effects Engine”* or similar, ***inside your game***.

CREDITS AND ACKNOWLEDGEMENTS

Effects Created By:

- Rhys Andrews
- Pim Schreurs
- Jake Wilson
- Joel Arnott.

Sprite/Graphic Work:

- Scott Llewelyn
- Pim Schreurs
- Rhys Andrews
- Joseph Chessey.

Special Thanks To:

- Joel Arnott
- Users @ GameMaker Community
- Users @ 64digits.com

CONTACT DETAILS

Website: www.gamecave.org
Email: administration@gamecave.org
GM-Community Topic: http://forums.gamemaker.nl/index.php?showtopic=138220
GM-Community PM: RhysAndrews

Please suggest any comments, bug reports, or questions via any of these contact methods. If you wish to state anything *publicly,* Please visit the **Gm-Community Topic**.