PyIgnition

User documentation

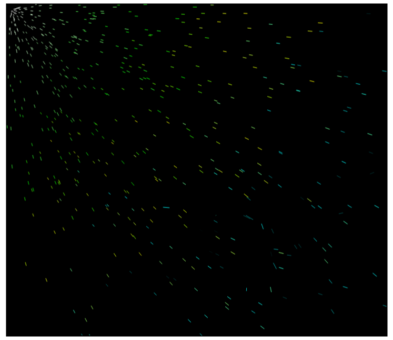
Version 1

Contents

Introduction

What is PyIgnition?

PyIgnition is an advanced particle effects engine written for use with Pygame, which can be used to generate particle effects for games and applications. It allows the user to create sources and other such objects through a central class which manages the entire effect, updating and redrawing all objects. It doesn't require its own mainloop, and can instead be called to update or to redraw at the user's command.



PyIgnition was conceived as a way of speeding up the often arduous process of creating particle effects in Pygame. At the time of writing Pygame has no built-in modules for such things, and although some external libraries do have features facilitating particle effect creation they are often fairly simple additions to larger game engines and do not receive much focussed development. The purpose of PyIgnition is to provide a complete library for the sole purpose of developing complex particle effects which can then be easily implemented in other projects.

Basics

How it works

As mentioned, all particles, sources and other objects for a particular effect come under the control of a `ParticleEffect` object. ParticleEffect objects represent completely separate particle effects – were you to create two, their effects would be handled completely discretely and would not interfere with each other in any way. Once initialised, `ParticleEffect` objects can be used to create instances of sources, gravities and obstacles (these cannot be created directly as they rely on a parent `ParticleEffect` class to manage them).

Creating the ParticleEffect object

ParticleEffect object initialisation

```
import PyIgnition
effect = PyIgnition.ParticleEffect(display,
position, size)

# Arguments:
     # display: the surface to draw to
     # position: the position of the effect
on-screen – used to shift an effect by
altering its 'pos' value
     # size: the size of the effect (normally
just the display size; currently has no effect
but may eventually be used for view clipping)
```

Saving and loading particle effects

PyIgnition `ParticleEffect` objects can also be stored in PPE ('PyIgnition Particle Effect') format, an XML-based format which lists all the objects the effect contains and enumerates their variables and keyframes. Saving and loading these files takes only a single line of code each way:

Saving a ParticleEffect object

```
effect.SaveToFile(filename)

# Arguments:
     # filename: the file path (either local
or absolute) of the file to save the effect in
```

Loading a ParticleEffect object

```
effect.LoadFromFile(filename)
# Arguments:
     # filename: the file path of the file
from which to load the particle effect
```

Calling a `ParticleEffect` object's `LoadFromFile()` function does not replace the objects it already holds, so it is relatively easy to combine multiple external effects files by loading them all into one `ParticleEffect` object.

Keyframing

Basics of keyframing

One of PyIgnition's most useful features is its versatile keyframing system. Almost every variable involved can be keyframed, and by doing so you can quickly build up elaborate animations.

In PyIgnition, keyframing is as simple as calling an object's `CreateKeyframe()` function and supplying the frame and the parameters to be keyframed. The specific parameters for each object will be listed in later sections, but for now here is a simple demo in which we keyframe a source to move from one place to another.

Keyframing a ParticleSource

```
source.CreateKeyframe(frame = 0, pos = (0, 0))
source.CreateKeyframe(frame = 30, pos = (50,
10))
```

This means that when the program runs, the source will move from (0, 0) to (50, 10) over the course of the first thirty frames.

Interpolation types

You can also specify which type of interpolation you would like a specific keyframe to use. PyIgnition currently supports two types of interpolation: linear (essentially moves from one value to another at a constant rate) and cosine (takes time to accelerate from one value and slows down as it approaches the next, which in many situations looks more natural). These can be specified for a particular keyframe simply by supplying the keyword argument 'interpolationtype' with a value of either "linear" or "cosine" (if no argument is supplied, linear is the default). The interpolation type needn't be constant for all frames in an animation, or even for all variables. For instance, in the following example we change a circle obstacle's colour linearly whilst simultaneously moving it from one place to another using cosine interpolation.

Mixing interpolation types

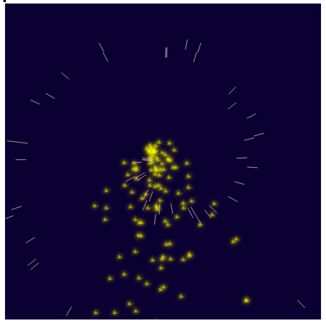
```
circle.CreateKeyframe(frame = 30, pos = (50, 
10), interpolationtype = "cosine")
circle.CreateKeyframe(frame = 40, colour = 
(255, 100, 163), interpolationtype = "linear")
```

Setting variables

There are some situations in which the normal keyframing method is inappropriate. For instance, if you wanted a particle source to follow the mouse cursor, you wouldn't be able to set up keyframes beforehand to accomplish this as the mouse's movement is entirely unpredictable. However, PyIgnition provides setter methods for all keyframeable variables which can be accessed using the `SetVariableName()` pattern. For instance, to set the particlesperframe variable of a `ParticleSource` object, you would call `source.SetParticlesPerFrame()`. (Note that function names follow the 'CamelCase' capitalisation style throughout.) This effectively creates a keyframe for said variable on the current frame, with the supplied value.

Consolidating keyframes

As you might imagine, doing this every frame over a long period of time will result in a fairly massive number of keyframes



being stored. This eventually causes decreased performance, and means that memory usage grows constantly over time. To avoid this, PyIgnition provides a `ConsolidateKeyframes()` function for all keyframeable objects which can be called simply by typing `objectname.ConsolidateKeyframes()`. This function deletes all past keyframes and creates a new keyframe on the present frame holding the current values for all variables. Called periodically, this can significantly improve performance both in programs which set variables every frames and ones which simply use more keyframes than usual.

Getting an object's current frame

In many situations it is helpful to be able to determine which frame of its keyframed animation an object is currently at – for instance, when you want to give an object a keyframe a certain number of frame in the future. Every keyframeable object in PyIgnition stores its current animation frame in a variable named `curframe`, which can be retrieved simply by accessing `object.curframe`.

Particles and sources

How particles are managed in PyIgnition

Source creates them, passes them up to the main object...

Creating and using a ParticleSource

The following example demonstrates how you would go about setting up a source to produce particles in PyIgnition. Not that the arguments have been omitted for now as there are very many of them; a full list can be found just after it.

Initialising a ParticleSource object

```
effect = ParticleEffect(screen, pos, size)
source = effect.CreateSource(arguments)
```

It is often wise to maintain a reference to the newly-created source (as demonstrated above) so that you can modify its parameters later or add keyframes to it. The full list of ParticleSource arguments is given below.

ParticleSource arguments

```
-pos (tuple): the position of the source
-initspeed (float): the initial speed of
spawned particles
-initdirection (float): the angle at which
particles are released (in radians)
```

```
-initspeedrandrange (float): the range (±
around the supplied initvelocity) of initial
speeds a particle could have
-initdirectionrandrange (float): as above, but
for the initial direction
-particlesperframe (int): the number of
particles to produce on each update cycle
-particlelife (int): the number of update
cycles for which a generated particle will
survive – supplying '-1' for this parameter
will make particles persist until the program
exits (not recommended unless your program
uses a limited number of particles)
-genspacing (int): the number of update cycles
to wait between generating particles (used to
make sources release particles in bursts
instead of constantly)
-drawtype (int): the drawtype to use for
particles (see the 'Particle parameters'
section below for a list)
-colour (tuple): the initial colour of
generated particles
-radius (float): the initial radius of
generated particles (only used by certain
drawtypes)
-length (float): the initial length of
generated particles (only used by certain
drawtypes)
-image (string): a path to the image to use
for generated particles (only used by certain
drawtypes)
```

Particle drawtypes and parameters

Particles can use any of the following drawtypes:

- `PyIgnition.DRAWTYPE_POINT` – draws simple point particles
- `PyIgnition.DRAWTYPE_CIRCLE` – draws particles as circles (uses `self.radius`)
- `PyIgnition.DRAWTYPE_LINE` – draws particles as lines of fixed length (uses `self.length`)
- `PyIgnition.DRAWTYPE_SCALELINE` – draws particles as lines which scale with their velocities
- `PyIgnition.DRAWTYPE_BUBBLE` – draws particles as unfilled circles or 'bubbles' (uses `self.radius`)
- `PyIgnition.DRAWTYPE_IMAGE` – draws particles as arbitrary images (uses `self.image`)

As you may have noticed, the three main parameters held by Particle objects are `radius`, `length` and `image`. You will probably never have to actually set these for individual particles; normally this would be done through the parent `ParticleSource` object by creating keyframes.

Keyframing

Both sources and the particles they produce can be keyframed through a reference to the source object:

ParticleSource keyframes

```
source.CreateKeyframe(args)
```

The arguments for this are the same as for creating a source, except `drawtype`, `colour`, `radius`, `length` and `image` are excluded as they belong to particles and not sources. Note that you do not have to supply a value for every variable; those which are not supplied will not be keyed for that frame. This way, you can effectively keyframe every variable individually.

Particle keyframes

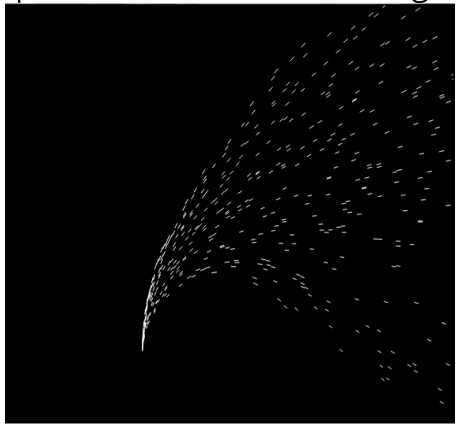
```
source.CreateParticleKeyframe(frame, colour,
radius, length, interpolationtype)

# Arguments:
     # colour (tuple) - the colour of the
particle
     # radius (float) - the radius of the
particle
     # length (float) - the length of the
particle
     # frame and interpolationtype - see
section 'Keyframing'
```

Note that a keyframe is automatically created on frame 0 using the colour, radius and length parameters specified for the source. This can of course be changed later in your program simply by using `CreateParticleKeyframe()` with 0 as the frame argument.

Gravities

How gravity works in PyIgnition

Gravities are essentially what they say on the tin: they are used to apply forces to particles in motion, accelerating them in particular directions. Each gravity object has its own



`GetForceOnParticle()` function, which calculates the force on a particle based on its position and its velocity. Behind the scenes, the `ParticleEffect` object loops through each particle and finds the sum of the forces being applied to it using all the existing gravity objects' `GetForceOnParticle()` functions. This total force is then used to accelerate the particle by simple application of Newton's second law.

PyIgnition allows you to create an unlimited number of gravity objects with each `ParticleEffect`, and gravities from one effect will have no influence on particles from a different effect, allowing you to have multiple completely separate particle effects running at once.

Types of gravity

PyIgnition currently supports three types of gravities: `PointGravity` is the most physically accurate as its strength follows an inverse square law with distance from its location; `DirectedGravity`, meanwhile, supplies constant gravitational pull in one direction, which is often more useful for normal purposes. `VortexGravity` throws realism to the wind, creating a massive whirlpool-like field which spirals particles around it whilst drawing them in. The following code examples show how to create each type of gravity.

Initialising a point gravity source

```
gravity = effect.CreatePointGravity(strength,
strengthrandrange, pos)

# Arguments:
     # strength (float) – essentially a
multiplier for the force calculated by
GetForce()
     # strengthrandrange (float) – similar to
the randrange variables supplied to
ParticleSources, this defines the range of
possible strength values on either side of the
supplied one.
     # pos (tuple) – the position of the point
gravity
```

Initialising a directed gravity source

```
gravity =
effect.CreateDirectedGravity(strength,
strengthrandrange, direction)

# Arguments:
     # strength and strengthrandrange – as
above
     # direction (tuple) – a vector defining
the direction of the gravity
```

Inisialising a votex gravity source

```
gravity = effect.CreateVortexGravity(strength,
strengthrandrange, pos)

# Arguments are the same as for PointGravity
objects
```

Keyframing

Much like particle sources, point gravities are keyframed through references returned by the creating functions (in each of the previous examples, this would be the 'gravity' object).

PointGravity and VortexGravity keyframes

```
gravity.CreateKeyframe(frame, strength,
strengthrandrange, pos, interpolationtype)
```

DirectedGravity keyframes

```
gravity.CreateKeyframe(frame, strength,
strengthrandrange, direction,
interpolationtype)
```

As you can see, the arguments when creating keyframes are the same as those for initialising the objects apart from the `frame` and `interpolationtype` parameters.

Other uses

Gravities in PyIgnition need not only be used for the suggested purpose of simulating gravity. They can also be used to direct particle flows, and when keyframed can help create interesting flow patterns. Also, by adding randomness to directed gravity you can simulate natural effects like wind which tend to fluctuate erratically.

Collisions with obstacles

Obstacle physics in PyIgnition

PyIgnition handles obstacles in much the same way as it handles gravities: they are created with a `ParticleEffect` object using the function `ParticleEffect.CreateObstacle(arguments)` (substituting in an obstacle type in place of 'obstacle'), and then modified using a returned reference. For example, here is how one would create a `Circle` obstacle:

Initialising a Circle obstacle

```
circle = effect.CreateCircle(arguments)
```

All obstacles take the tuple argument `pos` (which gives their starting position), the tuple argument `colour` (which specifies their draw colour) and the float argument `bounce` (which specifies how 'bouncy' the obstacle is – 0.5 is normally a good value for this, as 1.0 is often slightly too bouncy to look natural). Different types of obstacles will of course have their own additional arguments, and these are detailed in the 'types of obstacles' section below.

Types of obstacles

PyIgnition currently supports the following obstacle types:

➢ `Circle` – a circular obstacle, which takes the extra float paramater `radius`

➢ `Rectangle` – a rectangular obstacle, which takes the extra integer paramaters `width` and `height`

➢ `BoundaryLine` – a line which serves as a boundary for the effect (particles can only exist on one side of it), which takes the extra parameter `direction` (a tuple representing the normal vector)

Keyframing

Keyframing of obstacles is done in the same way as for all other objects – simply call an obstacle's `CreateKeyframe()` function, supplying a `frame` argument and the variables to set for that frame. For obstacles, all parameters are keyframeable.

Get involved

➢ Got an idea for a new feature or an improvement to the existing feature set? Submit it at blueprints.launchpad.net/pyignition!

➢ Found a bug? Submit a report to bugs.launchpad.net/pyignition and we'll fix it as soon as we can.

➢ If you have any general questions or wish to participate in development, feel free to contact me at animatinator@gmail.com.