How To Avoid Files Being Loaded in the Main Thread

BigWorld Technology 2.0. Released 2010.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Entity Preloads

When the client starts up, it will query each entity Python module for a function named `preload`. Resource names returned by this function will be loaded on client startup and kept in memory for the entire life-time of the client (i.e. it will be instantly available for use at any time). This is useful for commonly used assets to avoid potentially loading and re-loading at a later time. The tradeoff, however, is that the client will take longer to start and will use more memory (if the resource isn't actually being used at some point).

To use the preloads mechanism, create a global function called `preload` in the relevant entity module. It must take a single parameter which is a Python list containing resource to preload. Modify this list in place (e.g. using list.append or list concatenation), inserting the string names of each resource to be preloaded by the client.

For example,

```
# Avatar.py
import BigWorld

class Avatar( BigWorld.Entity ):
    def __init__( self ):
        ...

def preload( list ):
    list.append( "characters/avatars/human_avatar.model" )
    list.append( "system/maps/fx_flare_glow.tga" )
    ...
```

The type of resources which can be preloaded are,

- Fonts
- Textures
- Shaders
- Models

Chapter 2. `Entity.prerequisites()`

Assets may also be loaded from specific requests issued by the scripted logic. For instance, when the player equips a special item for the first time, that item's model and texture will have to be in memory before it can be displayed. In these cases, if that resource is not already resident, then the main thread will pause, waiting for the load request to complete.

Note

To prevent this kind of problems from creeping into the game, the BigWorld Client issues a warning in the debug output whenever an access to disk is requested from the main thread while the 3D scene is visible[1]. Programmers are encouraged to fix these warnings as soon as they start to pop-up.

The `prerequisites()` method is the recommended method of pre-loading the resources that may be required by the scripts in order to avoid pauses in the gameplay. It allows gameplay programmers to specify which assets can potentially be used by an entity instance. It will guarantee that the entity will not enter the world until all required resources have been loaded from disk, and are ready to be used by the main thread .

Note

In a truly dynamic game environment it is not possible to anticipate which resources will be requested by the scripts. In these cases, `BigWorld.loadResourceListBG()` can be used to load further resources.

Note

Although the entity entry into the world may be delayed, that usually is not noticeable, since most entities enter the player AoI at a great distance. Even when that is not the case (after teleporting to a location close to the player, for instance), that is still better than pausing the game or having avatars swing invisible swords

The function differs from “Preloads” by working on a per-entity instance basis, instead of globally. That allows a much more rational management of resources, since only assets with real potential of being used are stored in memory at any one time. Entity preloads on the other hand, are kept in memory for the whole life of the client application.

The fact that prerequisites work on a per-entity instance basis, as opposed to per-entity type, allows programmers to customize the prerequisites list depending on the state of the entity, further adding to the efficiency of the system. When a non-player entity enters the player AoI, for instance, only those items that the entity is currently carrying would be required to load. The Entity Manager system guarantees that the entity's properties are up-to-date before the prerequisites list is requested.

Note

For details on this function, see the Client Python API's entry **Class List → Entity**, section **Callback Method Documentation**.

[1] For details, see the Client Python API's entry **Modules → BigWorld**, section **Member Functions**, function **`BigWorld.worldDrawEnabled`**.

2.1. Using Entity.prerequisites()

The `Entity.prerequisites()` method is, in fact, very straightforward to use. After an entity is initialised, alongside its properties, but before the `onEnterWorld()` method is called, the `EntityManager` class calls the `prerequisites()` method on the entity. The method must return a list of all assets that must be loaded before the entity enters the world.

A simple example is illustrated below:

```
class TradeKiosk( BigWorld.Entity ):
  MODEL = 'models\kiosk.model'

  def prerequisites( self ):
    # return list even if it is just a single file
    return [ TradeKiosk.MODEL ]

  def onEnterWorld( self, prerequisites ):
    self.model = prerequisites[ TradeKiosk.MODEL ]
```

Simple example of loading prerequisite assets

A more realistic scenario is when the required resources vary according to the entity state. In this case, a single entity type is used to model all types of pickable flowers. The `type` property is used to differentiate each instance. Since `type` is guaranteed to be initialised when the `prerequisites` method is called, the script can customise which resources are loaded. In this case, the type property will not change over the lifetime of this particular entity, so we only need to load resource once.

```
class PickableFlower( BigWorld.Entity ):
  ASSETS = {
    'lili' : ('models\lili.model', 'effects\sparkles.xml')
    'daisy' : ('models\daisy.model', 'effects\spirits.xml')
    'orchid' : ('models\orchid.model', 'effects\petals.xml')
  }

  PICK_SOUND = 'sounds\pick_flower.wav'

  def prerequisites( self ):
    prereq = []
    prereq.append(PickableFlower.ASSETS[self.type][0])
    prereq.append(PickableFlower.ASSETS[self.type][1])
    prereq.append(PickableFlower.PICK_SOUND)
    return prereq

  def onEnterWorld( self, prerequisites ):
    # model varies according to the type of flower.
    # prerequisites can be accessed like a map, and returns python objects.
    # in this case, we know that .model files become PyModels.
    self.model = prerequisites[ PickableFlower.ASSETS[self.type][0] ]

    # it is up to us whether or not to hold onto the PyResourceRefs instance
    # passed into onEnterWorld.  In this case, we will be using the particle
    # system at a later date, so we'll simply hold onto the object until it
    # is needed.
    self.prerequisites = prerequisites

  def onPickEvent( self ):
    # picking sound effect is constant between types
    self.model.playSound( PickableFlower.PICK_SOUND )
```

```
    # particle effect varies according to the type of flower
    particles = self.prerequisites[ PickableFlower.ASSETS[self.type][1] ]
    self.model.root.attach( particles )
    particles.force(1)
```

Another example of load prerequisite assets

Chapter 3. Dynamically loading resources

There are times when you will need to dynamically load resources for an entity, instead of just when they enter the world. This might be due to a property change causing the entity's model to change, or perhaps the entity switches an item it is holding to one not yet seen before. In these cases, you will need to load and display a new model or other resource, and you will need load them in the background thread.

The `BigWorld.loadResourceListBG()` method works just like a dynamic version of `Entity.prerequisites`, it returns an object which existence ties the lifetime of the requested resources to itself. The actual resource loaded can be retrieved from the `PyResourceRefs` instance that is passed into the callback function. The types of Python resources that can be retrieved are:

- Models (type : PyModel)
- Particle Systems (type : PyMetaParticleSystem)
- Textures (type : PyTextureProvider)
- XML files (type : PyDataSection)
- FX files (type : PyMaterial)

The `BigWorld.fetchModel()` method can be used to asynchronously load just a `PyModel`. For details on this method, see the Client Python API.

The `Pixie.loadBG` method can be used to asynchronously load a `PyMetaParticleSystem`. For details on this method, see the Client Python API.

Additionally `BigWorld.loadResourceListBG()` can be used to hold onto C++ only resources, when you know that these are causing a problem. Here are the following types that can be loaded, and held onto, but not retrieved in Python:

- Visuals
- Lens Effects

3.1. Using `Entity.prerequisites` and `BigWorld.loadResourceListBG()`

The `BigWorld.loadResourceListBG()` method is useful for more complicated entities. For example, say you have an entity that can change its look at anytime to any of a large number of combinations. In this case, you can setup your initial look using prerequisites, but there is no way of knowing at which resources will be required in the future.

In order to implement this case, load the model asynchronously when needed, as in the following example:

```
class PickableFlower( BigWorld.Entity ):
  ASSETS = {
    'lili' : ('models\lili.model', 'effects\sparkles.xml')
    'daisy' : ('models\daisy.model', 'effects\spirits.xml')
    'orchid' : ('models\orchid.model', 'effects\petals.xml')
  }

  PICK_SOUND = 'sounds\pick_flower.wav'

  def __init__(self):
    pass
```

```
# This event handler is called after the entity is initialised,
# and its initial properties have been set.  We should return
# a list of resources that must be loaded into memory before
# onEnterWorld is called.
def prerequisite(self):
  prereqs = []
  prereqs += PickableFlower.PICK_SOUND
  prereqs += PickableFlower.ASSETS[self.type]
  return prereqs


# This event handler is called when the entity is first visible to
# the client, and when the prerequisite resources have been loaded.
# The prerequisites parameter is a PyResourceRefs instance, and
# can be used both to access the loaded resources, and to manage
# their lifetime.
def onEnterWorld(self, prerequisites):
  self.set_type(-1, prerequisites)


# This event handler is called whenever the entity type property
# is changed.  This can occur at anytime.  If we are calling this
# directly from onEnterWorld, the resources are already loaded and
# we can use them directly.  Otherwise we must queue up an
# asynchronous load of the required resources.
def set_type( self, old_type, prerequisites = None ):
  if self.type != old_type:
      if prerequisites == None:
          resourceList = PickableFlower.ASSETS[self.type]
          BigWorld.loadResourceListBG( resourceList, self.onLoad )
      else:
          self.onLoad( prerequisites )


# This user-defined callback function is called either when the
# loadResourceListBg method has finished, or is called by ourselves
# from enterWorld.
def onLoad( self, resourceRefs ):
  self.model = resourceRefs[ PickableFlower.ASSETS[self.type][0] ]
  self.particles = resourceRefs[ PickableFlower.ASSETS[self.type][1] ]
  self.model.root.attach( self.particles )


def onPickEvent(self):
  # picking sound effect is constant between types
  self.model.playSound(PickableFlower.PICK_SOUND)
  self.particles.force(1)
```

Asynchronously loading the model