awe6 – Tabula Rasa

General points (developers)

Properties

* Every writeable (non read-only) property has a setter.
* Every non-static property has a getter. Non-static properties might have their value set outside of the initialization procedure constructor / \_init. Whenever in doubt as to whether a property is non-static, add a getter.

Factory

The factory is the bootstrapping mechanism or the “entry point” for an awe6 application. The purpose of the factory is to spawn the kernel which serves as a repository for all other architectural components and to populate the kernel with these components. An awe6 application begins by instantiating a factory and its life cycle is complete when the spawned kernel is destroyed. Thus an awe6 application has precisely one factory and precisely one kernel. The factory is also the place where a number of different (constant) application properties are stored. Dynamic properties are kept in the kernel.

The basic duties of a factory are given in the following list.

1. Load the application configuration – from a file, from a website, from something else.
2. Initialize the application’s properties – author, application size, resource locations etc.
3. Launch the application’s kernel – it serves as a repository for the rest of the architectural components.
4. Populate the awe6 application’s kernel with the rest of the architectural components by injection.

The order is significant. A custom factory might perform more than that, but the above steps are necessary. The factory is responsible only for creating the architectural components. The kernel is then responsible for beginning and managing their life cycle.

Properties:

* application’s name
* application’s developer (company, for people see credits)
* application’s publisher
* application’s version (major, middle, minor, status designator alpha/beta…)
* application’s description short
* application’s description long
* application’s credits – format is Array<Array<Pair<Person, Duty>>>
* application’s background color – Int
* application’s encryption data (unknown type, Dynamic)
* supports resizing – does the application support dynamic resizing, e.g. by the mouse
* supports fullscreen / windowed
* application’s default mode (fullscreen / windowed)
* application’s default resolution
* application’s default bpp
* supported resolutions
* supported bit depths
* supports frame rate capping
* application’s ideal frame rate.
* application’s default screen type
* application’s config – the configuration is in JSON format allowing easier access to properties like myFactory.config.foo.boo.screenX.
* …
* application’s kernel

Functions:

* createAssetManager
* createInputManager
* createAudioManager
* createSceneManager
* …

No two awe6 applications are the same and therefore will require specific factories. The library should provide meaningful ready-to-use factories that should ease the development greatly.

# Kernel

The kernel of an awe6 application is responsible for the storage and management of all other architectural components except the factory.

The kernel is a tree of IKernelProcess instances. The root process is the kernel itself. The kernel is also viewable – the view is the main game view. All architectural components (owned) by the kernel are also processes. When a process is removed from the process tree all of its children are removed as well. A process is removed when its life cycle is complete or it’s explicitly disposed. The kernel is also an IView standing for the game’s screen and each added process which is also a view should be added to the view of the

Properties:

* factory
* assetManager
* inputManager
* soundManager
* sceneManager
* overlay
* application’s current size – this is the current size of the game
* application’s window size – this is the total area of the application
* application’s current bpp
* application’s current framerate
* application’s current memory usage
* application’s screen fitting mode
* fullscreen – is the application currently in fullscreen mode

## Kernel process

A kernel process is something that has a life cycle and lives inside the kernel. The activities of the kernel process could be influenced by many factors – mainly time, input, state of other processes, networking and randomness. All architectural components stored and managed by the kernel are kernel processes. Every kernel process has a process id and a parent. If the kernel process is displayable it should implement IViewable. If the kernel process has a view, the view is added to the view of the first displayable kernel process in the parent chain up to root. When the process is removed from the process tree, all of its children are removed and disposed of as well. Upon removal, the view of the process is also removed from the view tree.

Properties:

* parent – the parent process of this kernel process. This should be set by the kernel.
* Complete – whether the life cycle of this process has ended.
* age – the length of this process’ life cycle in seconds

Functions:

* update( timeDelta ) – notifies the process that the specified amount of time has elapsed since the last update. If the life cycle of the process is complete after this update or was complete, the method should return false.

# Scene Manager

The scene manager is the architectural responsible for managing the game’s (many) scenes and scene transitions. The scene manager defines for each pair of scene types the respective scene transition. Up to two scenes are managed at the same time: the current scene and the next scene when a scene transition is active.

## Scene

A scene describes a complete, independent part of the game. This will most often be a game screen like a main menu, a keyboard configuration screen, a video settings screen, select level screen, actual game etc. The scene is an entity container to hold all of the scene’s entities, but it’s not an entity itself.

## SceneTransition

The scene transition is a simple (or perhaps quite complex) graphical effect shown when the current scene is to be replaced by a new one.

Properties:

* onRemoveSceneCallback – the function that gets invoked when it’s save to remove the current scene.
* onRemoveSceneCallbackParams
* onAddSceneCallback – the function that gets invoked when it’s save to add the next scene
* onAddSceneCallbackParams
* onCompleteCallback – the function that gets invoked when the transition is complete
* onCompleteCallbackParams

# Input Manager

# Audio Manager

# IPausable

This interface must be implemented by classes that support pausing / unpausing logic. Typically this interface will be implemented by kernel processes that, when paused, will ignore updates.

Properties:

* paused – true if the object is currently paused, otherwise false
* pauserKey – the key used to pause the object, null if the object is not paused

Functions:

* pause(pauserKey) – pauses the object using the pauser key. If the object was successfully paused this method returns true; if the object is already paused, this function returns false.
* resume(pauserKey) – unpauses the object using the specified pauser key. If the was successfully resumed this method returns true; If the object was not paused or was paused by a different key, this method returns false.