

```

#ifndef __ResourceManager__
#define __ResourceManager__

#include <PA9.h>
#include <Assert.h>
#include "Error.h"

/* Resource Manager
 *
 * Resource life cycle:
 * Define a resource template
 * create resources from a template
 * Request a inactive resource
 * Return an active resource
 * delete resources
 *
 * eg sprites
 * Define how to load a sprite ONCE
 * Use this to create/load many.
 * Resources can be loaded once at the start
 * into a pool.
 * These can then be requested at runtime without
 * having to load into memory.
 *
 *
 */

#define RES_NULL      0x00
#define RES_LOADED   0x01
#define RES_ACTIVE    0x02

//typedef int T;
template<class T> class ResourceManager
//class ResourceManager
{
public:
    // Allocate arrays of pointers
    ResourceManager( u16 maxTemplates, u16 maxResources );
    virtual ~ResourceManager( void );

    // Define a resource from an example
    virtual u16 define( const T &r );

    // Returns a loaded resource from
    // a previously defined resource ID.
    // This resource will be created INACTIVE
    // and must be requested
    virtual void create( u16 templateId );
    // Creates N resources. They will be created INACTIVE
    // and must be accessed using the requiresResource()
    // function.
    virtual void create( u16 templateId, u16 number );

    // Request a loaded but currently unused resource.
    // Returns NULL if none are available.
    // Resources must be returned after use to be
    // reused
    virtual T* requestResource( u16 templateId );
    // After a resource has been requested, to
    // be used again, it must be returned at some point
    virtual void returnResource( u16 resourceId );
    // Return a resource by pointer
    // Slower than by ID as it requires a search
    virtual void returnResource( T* resource );

```

```

// Delete a resource.
// Free the memory and slot held by this resource.
// Which will allow more resources to be created.
virtual void free( u16 resourceId );
// Delete a resource referenced by pointer,
// Slower than by ID as it requires a search
virtual void free( T* resource );
// Delete a set number of resources created from
// a certain template. Will NOT free resources that
// are active. Will return the number of resources
// that it wasnt able to free because they were active.
// So if successful it will return 0
virtual u16 free( u16 templateId, u16 number );

// Here mainly for symmetry and for the deconstructor,
// so that templates may have depenancies defined outside
// this resource manager.
virtual void undefine( u16 templateId );

// DEFINED BY DERIVED CLASSES
// Updates resources.
// Called every frame, it carries out
// operations (defined by the derived class)
// on all resources.
virtual void updateResources();

T* getTemplatePtr( u16 templateId )
{
    if( templateId < maxTemplates ) return templates[ templateId ]; else return NULL;
}
T* getResourcePtr( u16 resourceId )
{
    if( resourceId < maxResources ) return resources[ resourceId ]; else return NULL;
}

u16 getTemplateNumber()
{
    return numTemplates; }
u16 getResourceNumber()
{
    return numResources; }
// Get number of resources created by a template
u16 getResourceNumber( u16 templateId );
// Get number of currently active resources
u16 getActiveResourceNumber();
u16 getActiveResourceNumber( u16 templateId );

T** templates;
u16 numTemplates;
u16 maxTemplates;

T** resources;
u16 numResources;
u16 maxResources;

// Array of resource flags,
// One u8 for each resource(matching indices)
// 0000 000X = Resource is loaded
// 0000 00X0 = Resource is active
u8* resourceFlags;

// linked to resources
// records the template used in its creation
u16* templateIds;
};

#include "ResourceManager.h"

/* this is to be removed once it has been tested, and
 * changed to a template
 */

```

```

template<class T>
ResourceManager<T>::ResourceManager( u16 maxTemplates, u16 maxResources )
{
    if( maxTemplates == 0 ||
        maxResources == 0 )
        throw Error("Zero max templates or maxResources is too low, in
ResourceManager<T>::ResourceManager()");
    // Init vars
    ResourceManager::numTemplates = 0;
    ResourceManager::maxTemplates = maxTemplates;

    ResourceManager::numResources = 0;
    ResourceManager::maxResources = maxResources;

    // Allocate Memory
    u16 i;
    // Template ptr array
    templates = new T*[ maxTemplates ];
    for( i=0; i<maxTemplates; i++ )
        templates[i] = NULL;

    // Resource ptr array
    resources = new T*[ maxResources ];
    for( i=0; i<maxResources; i++ )
        resources[i] = NULL;

    // Resource flags
    resourceFlags = new u8[ maxResources ];
    for( i=0; i<maxResources; i++ )
        resourceFlags[i] = 0x00;

    // templateIds
    templateIds = new u16[ maxResources ];
    for( i=0; i<maxResources; i++ )
        resourceFlags[i] = 0-1;
    // set as highest number
    possible to note that its not set.
}

template<class T>
ResourceManager<T>::~~ResourceManager( )
{
    u16 i;
    // All resources must be unloaded, as their
    // dependancies are undefined.
    for( i=0; i<maxResources; i++ )
        free(i);
    delete[] resources;
    delete[] resourceFlags;

    // Templates are not attached to anything, so they can
    // simply be deleted
    for( i=0; i<maxTemplates; i++ )
        undefine( i );
    delete[] templates;
}

template<class T>
u16 ResourceManager<T>::define( const T &r)
{
    if( numTemplates >= maxTemplates )
        throw Error("Max number of templates reached in ResourceManager<T>::define");

    // Find an empty slot for the template
    u16 id;
    for( id=0; id<maxTemplates; id++ )
        if( templates[id] == NULL )
            break;
    if( id >= maxTemplates )
        throw Error("No empty template slot exists in ResourceManager<T>::define");
}

```

```

        // Create the new template
        T *t = new T(r);

        // assign the new template
        templates[id] = t;
        numTemplates++;
        return id;
    }

template<class T>
void ResourceManager<T>::undefine( u16 templateId )
{
    assert( templateId <= maxTemplates &&
           templates[templateId] != NULL );

    delete templates[templateId];
    templates[ templateId ] = NULL;
    numTemplates--;
}

template<class T>
void ResourceManager<T>::create( u16 templateId )
{
    if( templateId < maxTemplates &&
        templates[ templateId ] != NULL )
    {
        // check there is a free slot
        if( numResources < maxResources )
        {
            // Find first available slot
            u16 id;
            for( id=0; id<maxResources; id++ )
                if( resources[id] == NULL )
                    break;

            // Create new resource from template
            resources[id] = new T( *templates[ templateId ] );
            resourceFlags[id] = RES_NULL;
            templateIds[id] = templateId;
            numResources++;

            return;
        }else
            throw Error("No free resource slot in ResourceManager::load()");
    }else
        throw Error("No template exists with that id in ResourceManager::load()");
}

template<class T>
void ResourceManager<T>::create( u16 templateId, u16 number)
{
    if( numResources+number < maxResources )
    {
        for( u8 i=0; i<number; i++ )
            create( templateId );
    }
    else
        throw Error("Not enough free resource slots to create resources in
ResourceManager<T>::create( u16 templateId, u16 number)");
}

template<class T>
void ResourceManager<T>::free( u16 resourceId )
{
    if( resourceId < maxResources )
    {
        if( resources[resourceId] != NULL )
        {

```

```

        delete resources[ resourceId ];
        resources[ resourceId ] = NULL;
        resourceFlags[ resourceId ] = RES_NULL;
        templateIds[ resourceId ] = 0-1;
        numResources--;
    }else
        throw Error("No resource exists at ResourceId,
ResourceManager::unload()");
    } else
        throw Error("ResourceId out of range in ResourceManager::unload()");
}

template<class T>
void ResourceManager<T>::free( T* resource )
{
    for( ul6 id=0; id < maxResources; id++ )
    {
        if( resources[id] == resource ){
            free( id );
            return;
        }
    }
    // else not found
    throw Error("Could not find resource in resources[], in
ResourceManager::returnResource()");
}

template<class T>
ul6 ResourceManager<T>::free( ul6 templateId, ul6 number )
{
    ul6 c=0,i;
    for( i=0; i<maxResources; i++ )
    {
        if( templateIds[i] == templateId )
            if( !(resourceFlags[i] & RES_ACTIVE) )
            {
                free( i );
                c++;
            }
    }
    // Return the number of resources unable to free
    // because they were active
    return number - c;
}

template<class T>
T* ResourceManager<T>::requestResource( ul6 templateId )
{
    // go through loaded resources find first inactive resource
    // which matches the template id and return a pointer to it.
    for( ul6 i=0; i<maxResources; i++ )
    {
        if( resources[ i ] != NULL && // if resource is created
            templateIds[i] == templateId && // if templates match
            (~resourceFlags[i] & RES_ACTIVE) ) // if flagged as inactive
        {
            resourceFlags[i] |= RES_ACTIVE; // set to active
            return resources[ i ]; // return teh resource
        }
    }
    // No inactive resource available
    return NULL;
}

template<class T>
void ResourceManager<T>::returnResource( ul6 resourceId )
{
    if( resourceId < maxResources &&
        resources[resourceId] != NULL )
    {

```

```

        resourceFlags[resourceId] &= (~RES_ACTIVE); // clear active bit
    }else
        throw Error("ResourceId is out of range, or not created, in
ResourceManager::returnResource()");
}

template<class T>
void ResourceManager<T>::returnResource( T* resource )
{
    for( ul6 id=0; id < maxResources; id++ )
    {
        if( resources[id] == resource ){
            returnResource( id );
            return;
        }
    }
    // else not found
    throw Error("Could not find resource in resources[], in
ResourceManager::returnResource()");
}

template<class T>
ul6 ResourceManager<T>::getResourceNumber( ul6 templateId )
{
    ul6 c=0,i;
    for( i=0; i<maxResources; i++ )
    {
        if( resources[i] != NULL &&
            templateIds[i] == templateId )
            c++;
    }
    return c;
}

template<class T>
ul6 ResourceManager<T>::getActiveResourceNumber()
{
    ul6 c=0, i;
    for( i=0; i<maxResources; i++ )
    {
        if( resources[i] != NULL &&
            resourceFlags[i] & RES_ACTIVE )
            c++;
    }
    return c;
}

template<class T>
ul6 ResourceManager<T>::getActiveResourceNumber( ul6 templateId )
{
    ul6 c=0, i;
    for( i=0; i<maxResources; i++ )
    {
        if( resources[i] != NULL &&
            templateIds[i] == templateId &&
            (resourceFlags[i] & RES_ACTIVE) )
            c++;
    }
    return c;
}

template<class T>
void ResourceManager<T>::updateResources()
{
}

#endif // __ResourceManager__

```