Tutorial for Application Life Cycle

- Startup and Asset Management
- Render Loop
- Finalization
- Asset Library Verification (Optional)
- Dynamic Properties
- Touch Support
- Loading of Shaped / Compressed Assets

Startup and Asset Management

Description

This tutorial leads through all required steps to initialize and prepare a Candera application for loading a SceneComposer generated asset and accessing content provided by the asset.

Initializing AssetLoader and Animations

AssetLoader

For loading assets and accessing content, the application needs to create an instance of an Candera::AssetProvider and Candera::ContentLoader to have them available for later usage.

```
    m_contentLoader = &ContentLoader::GetInstance();
    m_contentLoader->Init(&m_assetProvider);
```

Animations

If Animations are used in the scene, a time dispatcher for dispatching world time to animation players needs to be constructed during initialization.

```
    m_animationDispatcher = Animation::AnimationTimeDispatcher::Create();
    UpdateSystem* updateSystem = EntitySystem::Get<UpdateSystem>();
    if (0 != updateSystem) {
        UpdateSystem::Handle animationUpdateComponent = updateSystem->CreateComponent();
        CANDERA_SUPPRESS_LINT_FOR_SYMBOL(1025, Candera::UpdateDelegate::UpdateWithMilliseconds
, "False positive because the template arguments match.")
        UpdateSystem::Delegate animationUpdateDelegate = UpdateSystem::Delegate::UpdateWithMilli
            &Animation::AnimationTimeDispatcher::Update>();
        result = updateSystem->SetComponentUpdateDelegate(animationUpdateComponent, animationUpd
        result = updateSystem->AttachComponent(animationUpdateComponent, m_animationDispatcher.C
    }

#ifdef CANDERA_TRANSITIONS_ENABLED
    if (0 != m_transitionStrategy) {
```

```
        m_transitionStrategy->SetAnimationTimeDispatcher(m_animationDispatcher);
    }
#endif
```

Initializing 3D DefaultRenderMode

3D DefaultRenderMode

The default render mode is kind of a "global" render mode which is applied to all Nodes within a 3D Candera::Scene which have no separate Candera::RenderMode set. Further, if a Candera::Node has its own RenderMode set, still dedicated components can be derived from Candera::DefaultRenderMode - like e.g. CullMode - if the "inheritance bit" in the Node's associated RenderMode is set to true.

> If RenderMode is defined for a Candera::Camera, the RenderMode of the camera replaces the DefaultRenderMode, temporarily during the camera's render pass.

The DefaultRenderMode can be created and manipulated as follows:

```
    /* To overwrite the default render mode as specified
       in scene composer you can create your own render mode.
       This would have to be set via Renderer::SetDefaultRenderMode(renderMode);
     */
    Candera::MemoryManagement::SharedPointer<RenderMode> renderMode = RenderMode::Create();
    renderMode->SetWinding(RenderMode::CounterClockWise);
    renderMode->SetCulling(RenderMode::BackFaceCulling);
    renderMode->SetDepthWriteEnabled(true);
    renderMode->SetDepthTestEnabled(true);
```

Apply DefaultRenderMode

- In SceneComposer the default render mode can be configured in configuration menu - see Render Mode in SceneComposer User Manual
- In Candera, use the interface Candera::Renderer::SetDefaultRenderMode.

> When initializing an asset the default render mode in Candera will be replaced by the values from the asset!

Loading an Asset

Initializing AssetProvider

To initialize Candera::AssetProvider either a Candera::AssetRepository or Candera::AssetConfig has to be specified.

AssetRepository

Candera::AssetRepository is an abstraction of the actual location of the asset. Currently following implementations are available:

- Candera::FileAssetRepository for assets in file system
- Candera::MemoryAssetRepository for assets in memory, e.g. flash memory.

Custom AssetRepository implementations are possible.

AssetConfig

Candera::AssetConfig allows to specify multiple AssetRepository instances, typically for assets that are split with asset shaper and accessed on different locations. The application has to derive from the abstract Candera::AssetConfig to create an AssetConfig that supports the AssetRepository instances according to the specific needs. This AssetConfig can then be used to initialize Candera::AssetProvider:

```
    FEATSTD_LOG_INFO("- Initializing asset provider ...");

    static UInt32 validationFlags =
        AssetValidation::Flag<CffVersionAttribute, ErrorLevel>() |
        AssetValidation::Flag<FractionalNumberRepresentationAttribute, ErrorLevel>() |
        AssetValidation::Flag<PlatformNameAttribute, ErrorLevel>();

    if (!m_assetProvider.Initialize(&m_assetConfig, validationFlags)) {
        m_loadError = "Error: Asset initialization and validation failed!";
        FEATSTD_LOG_ERROR("%s\n", m_loadError);
        return false;
    }
```

Creating a FileAssetRepository

Before loading an asset from a file, ensure that the file exists at the given path. Then a Candera::FileAssetRepository can be created using the file path:

```
    FileAssetRepository* repo = CANDERA_NEW(FileAssetRepository)(fileName);
    FEATSTD_DEBUG_ASSERT(repo != 0);
```

If this operation succeeds, the asset repository can be added to an Candera::AssetConfig or specified directly to initialize Candera::AssetProvider.

Creating a MemoryAssetRepository

To create a Candera::MemoryAssetRepository the start address, where the asset can be found (e.g. the address in flash memory) has to be specified.

```
    MemoryAssetRepository* repo = CANDERA_NEW(MemoryAssetRepository)(startMemoryAddress);
    FEATSTD_DEBUG_ASSERT(repo != 0);
```

If this operation succeeds, the asset repository can be added to an Candera::AssetConfig or specified directly to initialize Candera::AssetProvider.

Retrieving AssetDescriptor

After initializing the Candera::AssetRepository, the Candera::AssetDescriptor can be retrieved:

```
    const AssetDescriptor& assetDescriptor = m_assetProvider.GetAssetDescriptor();
```

The AssetDescriptor contains information about the asset library file and Candera::AssetNamesList lists of all scenes, animations, displays etc. can be queried.

Create Displays and Render Targets

It is recommended to check if the asset contains at least one configured display, otherwise no display content will be visible in the application.

```
    if (assetDescriptor.GetAssetObjectCount(DisplayLib) == 0) {
        m_loadError = "Error: Missing elements.\n Please make sure that there are at least one c
        m_assetProvider.Finalize();
        return false;
    }
```

Next, the displays configured in the assets are created:

```
    Int displayId = -1;
    bool result = true;

    const AssetDescriptor& assetDescriptor = m_assetProvider.GetAssetDescriptor();

    Display* display = 0;
    Int displayWidth = -1;
```

```
    Int displayHeight = -1;
    for (AssetDescriptor::AssetIdIterator displayList = assetDescriptor.GetAssetIdIterator(
DisplayLib); displayList.IsValid(); ++displayList) {

        displayId = m_assetProvider.GetDisplayDataById(*displayList, displayWidth, displayHeight

        //check if already created before and destroy
        display = DevicePackageInterface::GetDisplay(displayId);
        if(display != 0) {
            DevicePackageInterface::DestroyDisplay(display);
        }
        display = DevicePackageInterface::CreateDisplay(displayId);
        if (display == 0) {
            FEATSTD_LOG_ERROR("Could not create display.\n");
            result = false;
        }
        static_cast<void>(m_displays.Add(display));

        // Load display meta information
        if (result) {
            if (!m_assetProvider.LoadDisplayProperties(*displayList)) {
                FEATSTD_LOG_INFO("- Loading of display properties failed");
            }
        }
        // Upload display if required
        if (result && upload) {
            Display::CommonSettings settings;
            settings.height = displayHeight;
            settings.width = displayWidth;
            if (!display->Upload(settings)) {
                FEATSTD_LOG_ERROR("Could not upload display.\n");
                result = false;
            }
        }
        // For simulation displays, set AutoKick disabled
        if ((display != 0)
            && (display->ToSimulatedDisplay() != 0)) {
                display->ToSimulatedDisplay()->SetAutoKickEnabled(false);
                FEATSTD_LOG_INFO("- AutoKick disabled");
        }
#ifdef FEATSTD_OS_WIN32
        //AppEnvironment::GetInstance().SetWindowPointer(display);
#endif
    }
    return result;
```

Here is an example how all render targets/layers can be read out of the asset file for further usage (like uploading, swapping, finalizing):

```
    GraphicDeviceUnit* gdu = 0;
    const AssetDescriptor& assetDescriptor = m_assetProvider.GetAssetDescriptor();
    Int rtCount = assetDescriptor.GetAssetObjectCount(RenderTargetLib);
```

```
    FEATSTD_UNUSED(rtCount);
    FEATSTD_LOG_INFO("Render target lib count = %d", rtCount);

#if (defined(CANDERA_3D_ENABLED) && defined(CGIAPP_STATISTICS_OVERLAY_ENABLED))
    static bool isRenderStatisticOverlayAttached = false;
#endif

    for (AssetDescriptor::RenderTargetIdIterator gduList = assetDescriptor.GetRenderTargetIdIter
        gdu = m_assetProvider.GetGraphicDeviceUnitById(*gduList);
        if (gdu != 0) {
            const Char* gduName = m_assetProvider.GetGraphicDeviceUnitById(*gduList)->GetName();
            FEATSTD_UNUSED(gduName);

#if (defined(CANDERA_3D_ENABLED) && defined(CGIAPP_STATISTICS_OVERLAY_ENABLED))
            Candera::DevicePackageDescriptor::UnitCategory unitCategory = DevicePackageDescriptc
            switch (unitCategory) {
                case DevicePackageDescriptor::DisplayTarget3D:
                case DevicePackageDescriptor::DisplayTarget2D:
                case DevicePackageDescriptor::Mixed2D3DDisplayTarget:
                    // attach listener to first valid RT
                    if (!isRenderStatisticOverlayAttached) {
                        Candera::RenderTarget2D* renderTarget = gdu->ToRenderTarget2D();

                        if (0 != renderTarget) {
                            renderTarget->AddEventListener(m_renderTargetEventListener);
                            isRenderStatisticOverlayAttached = true;
                        }
                    }
                    break;
                default:
                    //do nothing;
                    break;

            }
#endif
            FEATSTD_LOG_INFO("GDU name = %s", gduName);
            if (upload) {
                //only upload if specified
                if (gdu->Upload()) {
                    FEATSTD_LOG_INFO("GDU upload sucessful");
                }
                else {
                    FEATSTD_LOG_ERROR("Upload of GDU %s failed\n", gduName);
                }
            }
            static_cast<void>(m_gdus.Add(gdu));
        }
        else {
            FEATSTD_LOG_INFO("GDU null pointer returned");
        }
    }
```

Creating 3D Scenes from the Asset

Getting 3D Scene List

In case that Candera::AssetProvider has successfully loaded an asset file, 3D scenes part of the asset are constructed like in the following example.

By means of Candera::AssetDescriptor::GetAssetIdIterator, it is possible to retrieve a list of 3D Scenes:

```
    for (AssetDescriptor::AssetIdIterator sceneIdList = assetDescriptor.GetAssetIdIterator(
SceneLib); sceneIdList.IsValid(); ++sceneIdList) {
        static_cast<void>(sceneList.Add(m_assetProvider.GetSceneById(*sceneIdList)->GetScene()->
    }
```

Loading 3D Scene

The following listing shows how a Candera::Scene instance is finally loaded from the asset via Candera::ContentLoader::BuildSceneContext():

```
        // load scene by building the regarding scene context
        ContentLoader::UploadPolicy uploadPolicy = ContentLoader::NoVramUploadPolicy;
        if (upload) {
            uploadPolicy = ContentLoader::DefaultUploadPolicy;
        }

        ContentLoader::BuildState buildState = contentLoader->BuildSceneContextById(sceneId, upl

         // load all packages of the scene without interleaved rendering.
        while (buildState == ContentLoader::MorePackets) {
            buildState = contentLoader->BuildSceneContextById(sceneId, uploadPolicy);
        }
        if (buildState == ContentLoader::Completed) {
            sceneContext = contentLoader->GetSceneContextById(sceneId);
        } else {
            sceneContext = 0;
        }
```

The Candera::ContentLoader::DefaultUploadPolicy argument of the BuildSceneContext method forces the ContentLoader to create the scene data in the System RAM and to upload the asset data to VRAM. The upload of the asset data to VRAM can be avoided, if necessary, by using the argument Candera::ContentLoader::NoVramUploadPolicy instead.

The setting Candera::DefaultAssetProvider::SetIterativeLoadingEnabled() allows iterative loading of device objects during calls to Candera::ContentLoader::BuildSceneContext(). In combination with Candera::ContentLoader::NoVramUploadPolicy it is possible to load and upload a scene iteratively in background while another scene is rendered.

Further, all 3D Widgets related to the given scene can be retrieved from the SceneContext and configured as required. To support animated widgets, the application needs to provide a Candera::Animation::AnimationTimeDispatcher to its widgets, like this:

```
            Animation::AnimationTimeDispatcher::SharedPointer animationTimeDispatcher = GetAnima
            WidgetBase* widget = 0;
            for (bool success = sceneContext->GetFirstWidget(widget);
                success;
                success = sceneContext->GetNextWidget(widget)) {
                    if (widget != 0) {
                        widget->SetAnimationTimeDispatcher(animationTimeDispatcher);
                        // Register Controller for all GenericEventWidgets
                        ExampleWidgets::CgiEventWidget* gew = Dynamic_Cast<ExampleWidgets::CgiEv

                        if (gew != 0) {
                        #ifdef USE_STATE_MACHINE
                            // Register Statemachine for Widget Events
                            gew->Register(GenericController::UiEventCallback, 0);
                        #endif
                            // Set input event provider for every widget
                            gew->SetInputEventProvider(GetInputEventProvider());
                        // FEATSTD_LOG_INFO("* ++ widget registered for Input-Events ...\n");
                        }
                    }
            }

            for (TreeIterator<Node> nodeIterator(sceneContext->GetScene
()); nodeIterator.IsValid(); nodeIterator.Advance(TreeIterator<Node>::AdvanceToDescendant)) {
                CompositeGroup* compositeGroup = Dynamic_Cast<CompositeGroup*>(nodeIterator.GetN
                if (compositeGroup != 0) {
                    for (CompositeGroup::WidgetIterator widgetIterator = compositeGroup->GetWidg
                        widget = *widgetIterator;
                        widget->SetAnimationTimeDispatcher(animationTimeDispatcher);
                    }
                }
            }
```

Creating 2D Scenes from the Asset

Getting 2D Scene List

It is possible to retrieve a list of 2D Scenes from Candera::AssetDescriptor:

```
    for (AssetDescriptor::AssetIdIterator scene2DIdList = assetDescriptor.GetAssetIdIterator(
Scene2DLib); scene2DIdList.IsValid(); ++scene2DIdList) {
        static_cast<void>(sceneList.Add(m_assetProvider.GetScene2DById(*scene2DIdList)->GetScene
    }
```

Loading 2D Scene

The following listing shows how a Candera::Scene2D instance is finally loaded from the asset:

```
        // load scene by building the regarding scene context
        ContentLoader::UploadPolicy uploadPolicy = ContentLoader::NoVramUploadPolicy;
        if (upload) {
            uploadPolicy = ContentLoader::DefaultUploadPolicy;
        }

        ContentLoader::BuildState buildState = contentLoader->BuildScene2DContextById(sceneId, u

        // load all packages of the scene without interleaved rendering.
        while (buildState == ContentLoader::MorePackets) {
            buildState = contentLoader->BuildScene2DContextById(sceneId, uploadPolicy);
        }
        if (buildState == ContentLoader::Completed) {
            sceneContext = contentLoader->GetScene2DContextById(sceneId);
        } else {
            sceneContext = 0;
        }
```

As in the 3D case, the Candera::ContentLoader::DefaultUploadPolicy argument of the BuildScene2DContext method forces the ContentLoader to create the scene data in the System RAM and to upload the asset data to VRAM. The upload of the asset data to VRAM can be avoided, if necessary, using the argument Candera::ContentLoader::NoVramUploadPolicy instead.

The setting Candera::DefaultAssetProvider::SetIterativeLoadingEnabled() allows iterative loading of device objects during calls to Candera::ContentLoader::BuildSceneContext(). In combination with Candera::ContentLoader::NoVramUploadPolicy it is possible to load and upload a scene iteratively in background while another scene is rendered.

Again, similar to 3D widgets, 2D widgets related to the given scene can be retrieved from the SceneContext. Also in this 2D example the application provides a Candera::Animation::AnimationTimeDispatcher to its 2D widgets:

```
            Animation::AnimationTimeDispatcher::SharedPointer animationTimeDispatcher = GetAnima
            WidgetBase* widget = 0;
            for (bool success = sceneContext->GetFirstWidget(widget);
                success;
                success = sceneContext->GetNextWidget(widget)) {
                    if (widget != 0) {
```

```
                    widget->SetAnimationTimeDispatcher(animationTimeDispatcher);
                    // Register Controller for all GenericEventWidgets
                    ExampleWidgets::CgiEventWidget2D* gew = Dynamic_Cast<ExampleWidgets::Cgi

                    if (gew != 0) {
                    #ifdef USE_STATE_MACHINE
                        // Register Statemachine for Widget Events
                        gew->Register(GenericController::UiEventCallback, 0);
                    #endif
                        // Set input event provider for every widget
                        gew->SetInputEventProvider(GetInputEventProvider());
                    // FEATSTD_LOG_INFO("* ++ widget registered for Input-Events ...\n");
                    }
                }
            }



            for (TreeIterator<Node2D> nodeIterator(sceneContext->GetScene
()); nodeIterator.IsValid(); nodeIterator.Advance(TreeIterator<Node2D>::AdvanceToDescendant)) {
                CompositeGroup2D* compositeGroup = Dynamic_Cast<CompositeGroup2D*>(nodeIterator.
                if (compositeGroup != 0) {
                    for (CompositeGroup2D::WidgetIterator widgetIterator = compositeGroup->GetWi
                        widget = *widgetIterator;
                        widget->SetAnimationTimeDispatcher(animationTimeDispatcher);
                    }
                }
            }
```

Loading Animation an AnimationGroup

Loading Animation

As can be seen from application initialization, the application already configured a Candera::AnimationTimeDispatcher.

Animations configured in SceneComposer can be loaded from the asset into a Candera::AnimationPlayer and started like following:

```
        Animation::AnimationPlayerBase::SharedPointer animationPlayer = m_assetProvider.GetAnima
        if (animationPlayer != 0) {
            static_cast<void>(m_animationDispatcher->AddPlayer(animationPlayer));
            if (!m_animationPlayers.Contains(animationPlayer)) {
                static_cast<void>(m_animationPlayers.Add(animationPlayer));
            }
            if (!animationPlayer->Start()) {
                FEATSTD_LOG_ERROR(" ERROR: animation player start failed for'%s\n", animationNam
            }
        }
```

Loading AnimationGroup

Candera::AnimationGroup allows to start a hierarchical group of animations as configured in SceneComposer. The steps are similar besides that all children within the group have to be added to the Candera::AnimationTimeDispatcher, so it is advised to iterate through the tree recursively adding each child. The group can then be started at once:

```
        if (animationPlayer->IsTypeOf(Animation::AnimationGroupPlayer::GetTypeId())) {

            Animation::AnimationGroupPlayer::SharedPointer animationGroupPlayer
                = Dynamic_Cast<Animation::AnimationGroupPlayer::SharedPointer>(animationPlayer);
            if (animationGroupPlayer != 0) {
                AddAnimationGroupChildren(animationGroupPlayer,
                    animationGroupPlayer->GetFirstSuccessor(SharedPointer<Animation::AnimationPl
                if (!animationGroupPlayer->Start()) {
                    FEATSTD_LOG_ERROR(" ERROR: animation player group start failed for'%s\n", ar
                }
            }
```

Selecting a Theme

Retrieving Available Themes

The available themes in the asset can be retrieved via Candera::AssetDescriptor:

```
    for (AssetDescriptor::AssetIdIterator themeIdList = assetDescriptor.GetAssetIdIterator(
ThemeLib); themeIdList.IsValid(); ++themeIdList) {
        static_cast<void>(themeList.Add(*themeIdList));
    }
```

Selecting a Theme

Afterwards one of the available themes can be set to Candera::AssetProvider:

```
            Candera::Id themeId = AssetProviderFunctions::GetIdByName(&m_assetProvider,
ThemeLib, themeName);
            m_assetProvider.SetCurrentThemeById(themeId);
```

From now on this theme will be used when loading new scenes.

> You need to reload the scene explicitly when you change the theme!

Loading User Data

User data are raw data of any arbitrary type or format.

A SceneComposer asset can be used as a pass-through for such kind of proprietary data to the application.

Steps required:

- Import Raw Data to the SceneComposer solution
- Assign a numerical asset id like "1" to the data item in SceneComposer, optionally assign a symbolic name for this item
- Export the asset (also the symbolic names header file is exported) and load it in the application
- Access the raw data via a ResourceHandle retrieved from Candera::AssetProvider based on the Asset ID(see below)

```
    Id jpgId = CDA_LIBRARY_ASSETID(0x01); // Instead, a generated symbolic name could be used
    ResourceHandle resH = m_assetProvider->OpenRawResourceById(jpgId);
    Int32 resSize = m_assetProvider->GetResourceSize(resH);
    const void* dataPointer = m_assetProvider->GetResourceAddress(resH);
    UInt8* buffer = CANDERA_NEW_ARRAY(UInt8, resSize);
    UInt32 readBytes = m_assetProvider->ReadResource(resH, buffer, 0, resSize);
    CANDERA_DEBUG_ASSERT(resSize == readBytes);
    m_assetProvider->CloseResource(resH);
    //...
    CANDERA_DELETE_ARRAY(buffer);
```

Accessing Items via Asset Id

Item Identification: Asset Id instead of String

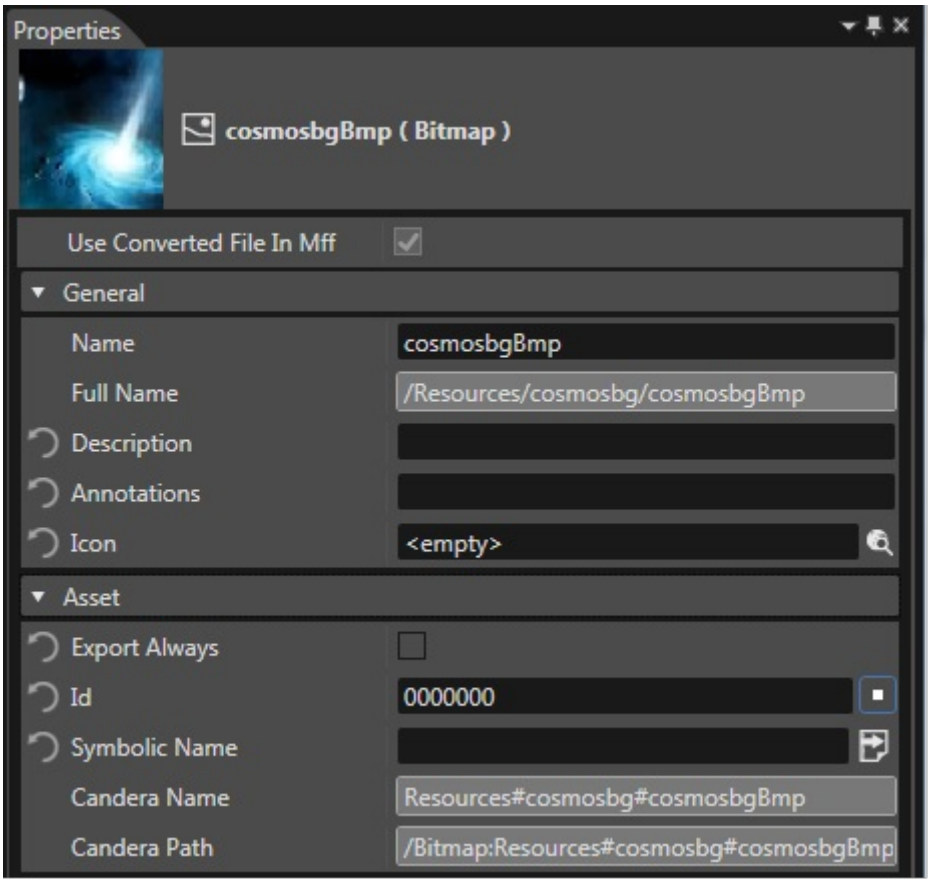
All items in a solution have a type and a name which identifies them in their parent container and a full name which uniquely identifies them in the entire solution. For example, a mesh in a scene could have the full name `/Global/Scenes/MyScene/Group:RootNode/Mesh:MyMesh`.

Before CGI-Studio V3.0.0, for accessing items from an application, the `CanderaPath` property displayed the string, which uniquely identified this item in the asset library. For the example above, the same mesh would have been accessed from Candera via the name `/Scene:Global::Scenes::MyScene/Group:RootNode/Mesh:MyMesh`.

To improve asset loading performance, CGI-Studio V3.0.0 introduces numerical Ids for accessing items from asset libraries.

Using Asset Ids

As an example, a bitmap called "cosmosbgBmp" is assigned a symbolic name in a SceneComposer solution:



The "Id" field in SceneComposer displays by default "0000000". For this default, SceneComposer will generate a hash value for the Id. In case a custom value greater than '0' is entered in this field, the custom value will be generated instead, which is guaranteed to remain the same permanently, while the generated hash value could change over time.

Either during asset export, or explicitly via 'File -> Export Symbolic Names ...' menu, a header file is generated by SceneComposer containing all symbolic names defined in the solution. For the bitmap symbolic names, the export looks like:

```
namespace CgiAssetNames {
    static const Candera::Id cosmosbgBmp = CDA_LIBRARY_ASSETID(0x0U, 0x11008CU);
} //namespace CgiAssetNames
```

To access this bitmap, use the interface Candera::AssetProvider::GetBitmapById:

```
        Bitmap::SharedPointer bitmap = Base::GetAssetProvider()->GetBitmapById(CgiAssetNames::cc
```

Use the same approach for all other assets like displays, render target, nodes, font and raw data, ...

Query Candera Compile Switches

Candera Compile Switches

Note that Candera compile switches are documented in Candera Configuration.

As an example, in case of DEVICE device, following queries are supported: (DEVICE stands for your device)

```
#if CANDERA_DEVICE==DEVICE
    ...
#endif
```

Similarily:

```
#ifdef CANDERA_DEVICE_DEVICE
    ...
#endif
```

Render Loop

Description

After loading an asset and creating scenes, this chapter will now explain how an application can trigger and control rendering of 3D and 2D scenes.

Basically it is up to the application to define an event loop triggering render calls. Candera provides interfaces to render specific or all cameras, but scheduling and triggering these render calls is what the application is responsible for.

3D Render Loop

3D Render Calls

Static class Candera::Renderer is part of Candera 3D Engine and provides two render call interfaces:

1. Render all 3D cameras: Candera::Renderer::RenderAllCameras
2. Render a specific camera only: Candera::Renderer::RenderCamera

Usually, rendering of all cameras is the safer and most convenient way, hence chosen here for the tutorial:

```
    FEATSTD_LINT_NEXT_EXPRESSION(534, "return value not needed");
    Renderer::RenderAllCameras();
```

Render Buffer Swapping

If the draw target operates on multiple buffers after each rendering cycle the presentation buffer is exchanged with the current background buffer, on which was written during the cycle.

For this, three interfaces are available:

- Candera::RenderTarget::SetAutoSwapEnabled(bool enabled): if *true*, render target buffers are swapped automatically, when a camera within the render target has been rendered. Default: *false*

- Candera::Camera::SetSwapEnabled(bool enable): if *true*, buffers are swapped automatically after rendering the according camera. Default: *false*
- Candera::RenderTarget::SwapBuffers(): Trigger buffer swapping 'by hand'

It is very important to control buffer swapping in the application correctly:

- If swapping is never done, the display will stay black
- If buffers are swapped too often, the display will flicker.

The application doesn't need to swap its render targets in case:

- the render target is set to swap automatically. For this, set the render target property "SetAutoSwapEnabled" to *true*.
- the proper camera is set to swap automatically. For this, set the camera property "SwapEnabled" to *true*.

> If you use more than one camera on a render target, take care to enable swapping only for the last camera (highest sequence number).

2D Render Loop

2D Render Calls

2D Render Call interfaces are provided by Candera::Renderer2D.

This is the 2D specific render call equivalent to 3D.

```
    Renderer2D::RenderAllCameras();
```

Render Buffer Swapping

For 2D, render buffer swapping is analogous to 3D: The default value of 2D RenderTarget property "AutoSwapEnabled" and 2D Camera property "SwapEnabled" is *false*.

So take care to

- either swap render target
- or swap one of the cameras used
- or trigger buffer swapping on the render target 'by hand'.

Finalization

Description

This tutorial explains how to shut down a 2D or 3D application safely.

Releasing Scenes

First all 3D or 2D scenes must be released.

- 3D Scenes:

```
    contentLoader->ReleaseAllSceneContexts();
    m_sceneContexts.Clear();
```

- 2D Scenes: The same procedure as for 3D Scenes, whereas the Candera::ContentLoader interface for releasing 2D scene contexts is different:

```
    contentLoader->ReleaseAllScene2DContexts();
    m_sceneContexts.Clear();
```

Don't forget to shutdown the TextEngine, if there is one in use:
`Candera::TextRendering::FontEngine::Instance().Shutdown();`

Releasing AnimationPlayer

Next, all used Candera::AnimationPlayer and Candera::AnimationPlayerGroup instances have to be removed from the Candera::AnimationTimeDispatcher and freed.

Unloading RenderTargets and Displays

All involved render targets / layers need to be unloaded from video memory:

```
    if (m_isRenderTargetsUploaded) {
        for (Int index = FeatStd::Internal::NumericConversion<Int>(m_gdus.Size()) - 1; index >=
            GraphicDeviceUnit* gdu = m_gdus[index];
            if (gdu != 0) {
                if (0 != gdu->ToRenderTarget3D()) {
                    static_cast<void>(gdu->ToRenderTarget3D()->Activate());
                }
```

```
                gdu->Unload();
            }
            m_gdus[index] = 0;
        }
    }
    m_gdus.Clear();
```

Destory Displays in use:

```
    FEATSTD_LOG_INFO("- Destroy displays ...");
    Vector<Display*>::Iterator it;
    for (it = m_displays.Begin(); it != m_displays.End(); ++it) {
        Display* display = *it;
        if (display != 0) {
            display->Unload();
            DevicePackageInterface::DestroyDisplay(display);
        }
    }
    FEATSTD_LOG_INFO("- done");
    m_displays.Clear();
```

Finalizing AssetProvider

Finally, Candera::AssetProvider and Candera::ContentLoader shall be finalized:

```
    m_assetProvider.Finalize();
    m_contentLoader = 0;
```

Destructing Other Resources

Don't forget to destruct any other resources in the application destructor.

Asset Library Verification (Optional)

Description

The optional Asset Library Verification feature allows asset version checks and provides means to verify the compatibility between an asset library file and an application.

Asset Version Information

The asset file contains the following data to support asset verification:

- **Asset Version (Int32):** Describes the version of the asset file format. This version must match the version of the Candera AssetLoader to be able to load the asset file.
- **File Size (Int32):** Size of the file in bytes.
- **File Timestamp (Int64):** Contains the creation time of the asset.
- **Widget Hash (UInt32):** Hash value generated from the available widgets and their properties.
- **Widget Set Version (Int32):** Version of the Widget Set.
- **Candera Version (Int32[4]):** Candera version used within SceneComposer.
- **Solution GUID (Char[16]):** Unique ID of the Solution.
- **Custom ID (Int32):** Custom Id that can be set by the user upon asset generation.
- **SceneComposer Version (Int32[4]):** SceneComposer version used to generate the asset file.

Feature Comparison

Without the optional asset verification, only a limited set of information is available. The table below gives a comparison of asset information available by default in Candera, and asset information available with enabled asset verification feature.

	Candera Default	Asset Verification Enabled
Asset Version	X	
File Size	X	
Time Stamp		X

Widget Hash		X
Widget Set Version		X
Candera Version		X
Solution GUID		X
Custom ID		X
SceneComposer Version		X

Export of Asset Version Info

Automatic / User Defined Values

Most of the asset information is automatically generated by SceneComposer during asset export, but some can be controlled directly. The following table shows which properties can be set manually.

	User Defined	SceneComposer
Asset Version		X
File Size		X
Time Stamp		X
Widget Hash		X
Widget Set Version	X	
Candera Version		X
Solution GUID		X
Custom ID	X	
Scenecomposer Version		X

Set the WidgetSet Version Number

To set the user-defined WidgetSet version number the new macro `CdaVersion` can be used in the WidgetSet definition.

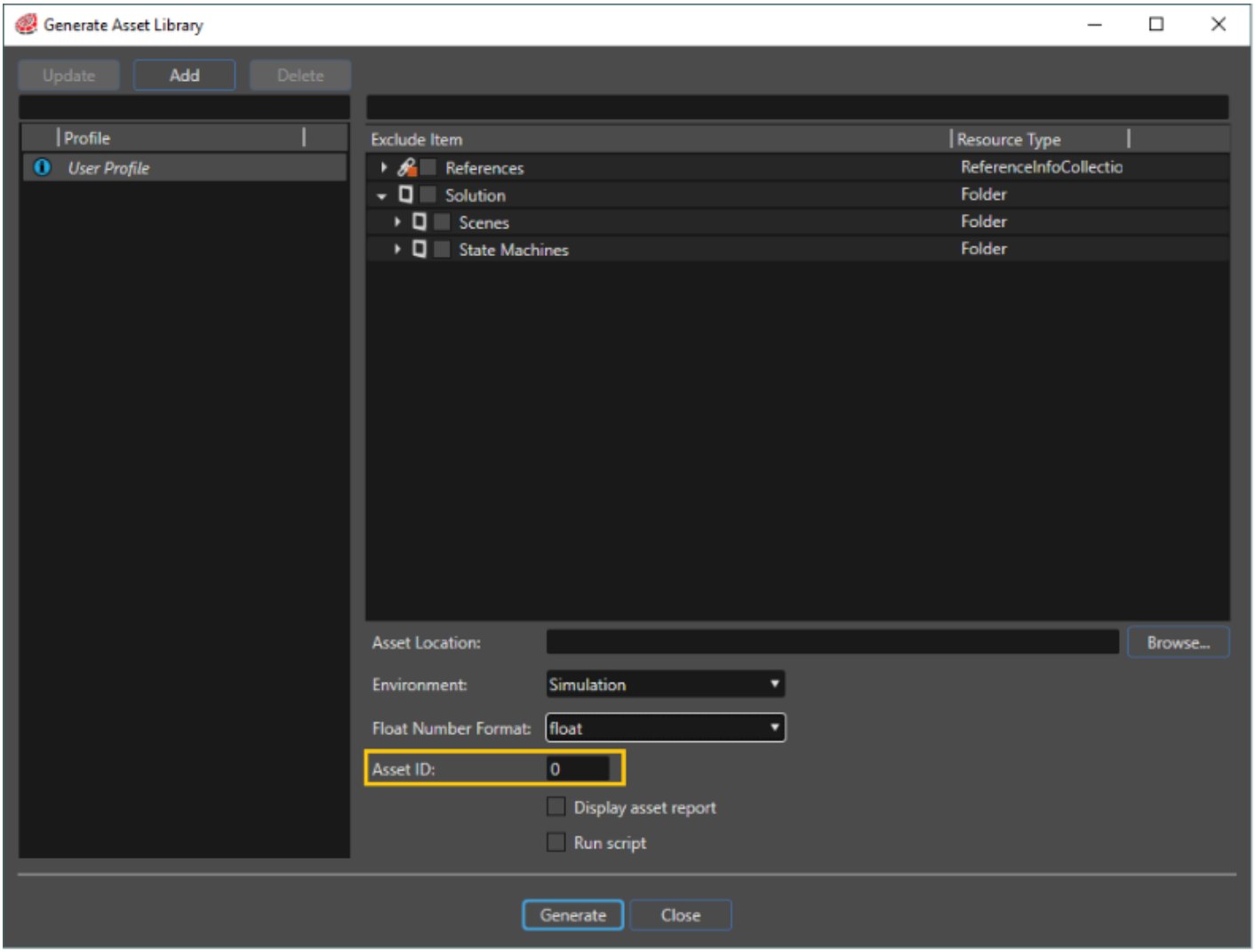
```
CdaWidgetSet(DemoWidgetSet)
    CdaDescription("Widget Set Demo")
    CdaVersion(1) // Sets the Widget-Set version
    CdaWidgets()
```

```
    CdaWidget(DemoWidget1)
    CdaWidget(DemoWidget2)

    CdaWidgetsEnd()
CdaWidgetSetEnd()
```

Set the Custom-Id

The Custom-Id can be set inside of the asset creation dialog of SceneComposer.



Asset Verification at Runtime

Read Asset Version Data from Asset

The following code example will show how to extract the asset version information from an Asset Repository.

```
    AssetData::AssetVersionInfo versionInfo;
    m_assetDescriptor->GetVersionInfo(versionInfo);
```

The information is now available in the `versionInfo` structure.

Get Runtime Version Info from Application

To compare asset version info from the loaded asset with the runtime values from the application, use following interfaces:

- Get Candera Version Number
- Get the Supported Asset Version
- WidgetSet verification

Get Candera Version Number

The version number of the currently used Candera can be found in the `CanderaVersionGen.h` file. This file is generated automatically by the build system and contains defines similar to this, describing the exact version of Candera.

```
#define CANDERA_VERSION_MAJOR 3
#define CANDERA_VERSION_MINOR 12
#define CANDERA_VERSION_PATCH 0
#define CANDERA_VERSION_TWEAK 0
```

And can be used for example like this:

```
    AssetData::AssetVersionInfo versionInfo;
    assetDescriptor->GetVersionInfo(versionInfo);
    FEATSTD_LOG_INFO("Current Candera version = %d.%d.%d.%d, asset generated with version =  %d.
        CANDERA_VERSION_MAJOR, CANDERA_VERSION_MINOR, CANDERA_VERSION_PATCH, CANDERA_VERSION_TWE
        versionInfo.m_canderaVersion[0],versionInfo.m_canderaVersion[1],versionInfo.m_canderaVer
```

Get the Supported Asset Version

The number of the asset file format that is supported by the asset loader is located in the `AssetValidation.h` file and can be accessed like this:

```
    if (versionInfo.m_fileVersion != Candera::AssetValidation::CURRENT_VERSION) {
        FEATSTD_LOG_ERROR("Asset version (%d) does not match supported asset file version (%d)\r
    }
```

WidgetSet verification

To retrieve the currently used WidgetSet version number and WidgetSet hash value, the following code can be used.

```
    if (GetWidgetSet() != 0) {
        FEATSTD_LOG_INFO("Widget set version: %d, asset generated with version: %d",
```

```
            GetWidgetSet()->GetVersion(), versionInfo.m_widgetSetVersion);
        FEATSTD_LOG_INFO("Widget set hash: %d, asset generated with hash: %d",
            GetWidgetSet()->GetHash(), versionInfo.m_widgetHash);
    }
```

> The `GetWidgetSet`-Function is part of the `WidgetMetaInfo.h` file.

Candera::AssetValidation

Another option of asset verification is to use Candera::AssetValidation during asset loading. In this case, specific asset validation flags will be evaluated during asset loading

Validation Attributes

Refer to Candera::AssetValidationAttributes for all attributes which can be used for asset validation during asset loading.

Validation Levels

Refer to Candera::AssetValidationLevels for the actions, which shall be taken after asset validation.

Asset Validation during Asset Loading

Candera::DefaultAssetProvider::Initialize method allows to specify asset validation flags as a parameter:

```
    if (!m_assetProvider->Initialize(&m_assetConfig, m_validationFlags)) {
        m_assetConfig.ClearRepositoryList();
        return false;
    }
```

Dynamic Properties

How to attach a Dynamic Property to CanderaObject

An application may want to attach any kind of application specific data to an arbitrary CanderaObject. This is easily possible as CanderaObject derives from DynamicPropertyHost, which can store dynamic properties. Different to conventional object properties, a dynamic property can be attached to an object, without being declared in the object itself. Further, a dynamic property does not allocate memory unless its value is set explicitly and differs from its default value.

Find below a simple example how to attach an application specific object, acting as dynamic property, to an object that derives from DynamicPropertyHost, like e.g. CanderaObject, Node, Mesh, Appearance, Texture, only to name a few.

- If possible the dynamic property definition (see class LayouterDynamicProperties) should be contained in the source file and not in the header file. Otherwise, the code size may get unnecessarily increased due to the inability of Gcc and Multi linker to get rid of duplicate template instantiation in the object files.
- Dynamic properties
  - can be set (see SetSize and SetLayoutDirection)
  - can be retrieved (see GetSize and GetLayoutDirection)
  - can be checked if they are set (see IsSizeSet and IsLayoutDirectionSet)
  - can be cleared (see ClearLayoutDirection)
  - have default values: C++ default initialization for simple types (see LayoutDirection) or specific values for complex types (see Size)
  - can have an optional change notification (see OnSizeChanged and OnLayoutDirectionChanged)
- A convenience interface should be provided to set, get, clear and check the dynamic property (see class Layouter). The dynamic property is only accessed via that interface.
- The DynamicPropertyHost (see class Layouter) has to provide the ParentProvider method for inherited dynamic properties.
  - In the Layouter sample it is guaranteed by the Layouter convenience interface that only CanderaObject instances are supported. Hence, a static_cast to CanderaObject is safe.

- Layouter.h

```
class Layouter : public CanderaObject
{
    …
    static void SetSize(CanderaObject& node, const Vector2& size);
    static const Vector2& GetSize(const CanderaObject& node);
```

```
    static bool IsSizeSet(const CanderaObject &node);
    …
    static void SetLayoutDirection(CanderaObject& node,
LayoutAlignment::LayoutDirection::Enum direction);
    static LayoutAlignment::LayoutDirection::Enum
 GetLayoutDirection(const CanderaObject& node);
    static bool IsLayoutDirectionSet(const CanderaObject &node);
    static void ClearLayoutDirection(CanderaObject& node);
    …
    static const Candera::DynamicProperties::DynamicPropertyHost
* ParentProvider(const Candera::DynamicProperties::DynamicPropertyHost* host);
    …
    static const Vector2& SizeDefault();
    …
    static void OnSizeChanged(DynamicPropertyHost* obj, const DynamicProperties::ValueChang
    …
    static void OnLayoutDirectionChanged(DynamicPropertyHost* obj, const DynamicProperties:
    …
};
```

- Layouter.cpp

```
class LayouterDynamicProperties
{
    …
    static void SetSize(CanderaObject& node, const Vector2& size)
    {
        static_cast<void>(node.SetValue(CdaDynamicPropertyInstance(Size), size));
    }

    static const Vector2& GetSize(const CanderaObject& node)
    {
        return node.GetValue(CdaDynamicPropertyInstance(Size));
    }

    static bool IsSizeSet(const CanderaObject &node)
    {
        return node.IsValueSet(CdaDynamicPropertyInstance(Size));
    }
    …
    static void SetLayoutDirection(CanderaObject& node,
LayoutAlignment::LayoutDirection::Enum direction)
    {
        static_cast<void>(node.SetValue(CdaDynamicPropertyInstance
(LayoutDirection), direction));
    }

    static LayoutAlignment::LayoutDirection::Enum
```

```
 GetLayoutDirection(const CanderaObject& node)
    {
        return node.GetValue(CdaDynamicPropertyInstance(LayoutDirection));
    }

    static bool IsLayoutDirectionSet(const CanderaObject& node)
    {
        return node.IsValueSet(CdaDynamicPropertyInstance(LayoutDirection));
    }

    static void ClearLayoutDirection(CanderaObject& node)
    {
        static_cast<void>(node.ClearValue(CdaDynamicPropertyInstance
(LayoutDirection)));
    }
    …
    CdaDynamicPropertiesDefinition(Candera::Layouter, CanderaObject);
        …
        CdaDynamicProperty(Size, Vector2);
            CdaDynamicPropertyValueChangedCb(&Layouter::OnSizeChanged);
            CdaDynamicPropertyDefaultValue(Layouter::SizeDefault());
            …
        CdaDynamicPropertyEnd();
        …
        CdaDynamicProperty(LayoutDirection, LayoutAlignment::LayoutDirection::Enum);
            CdaDynamicPropertyValueChangedCb(&Layouter::OnLayoutDirectionChanged);
            …
        CdaDynamicPropertyEnd();
        …
    CdaDynamicPropertiesEnd();
};
…
void Layouter::SetSize(CanderaObject& node, const Vector2& size)
{
    LayouterDynamicProperties::SetSize(node, size);
}

const Vector2& Layouter::GetSize(const CanderaObject& node)
{
    return LayouterDynamicProperties::GetSize(node);
}

bool Layouter::IsSizeSet(const CanderaObject &node)
{
    return LayouterDynamicProperties::IsSizeSet(node);
}
…
void Layouter::SetLayoutDirection(CanderaObject& node,
```

```
LayoutAlignment::LayoutDirection::Enum direction)
{
    LayouterDynamicProperties::SetLayoutDirection(node, direction);
}

LayoutAlignment::LayoutDirection::Enum
 Layouter::GetLayoutDirection(const CanderaObject& node)
{
    return LayouterDynamicProperties::GetLayoutDirection(node);
}

bool Layouter::IsLayoutDirectionSet(const CanderaObject& node)
{
    return LayouterDynamicProperties::IsLayoutDirectionSet(node);
}

void Layouter::ClearLayoutDirection(CanderaObject& node)
{
    LayouterDynamicProperties::ClearLayoutDirection(node);
}
…
const Candera::DynamicProperties::DynamicPropertyHost* Layouter::ParentProvider(const
Candera::DynamicProperties::DynamicPropertyHost* host) {
    CANDERA_SUPPRESS_LINT_FOR_NEXT_EXPRESSION(1774, "Layouter interface only allows Candera
    const CanderaObject* object = static_cast<const CanderaObject*>(host);
#ifdef CANDERA_2D_ENABLED
    const Node2D* node2D = Dynamic_Cast<const Node2D*>(object);
    if (0 != node2D) {
        return node2D->GetParent();
    }
#endif
#ifdef CANDERA_3D_ENABLED
    const Node* node = Dynamic_Cast<const Node*>(object);
    if (0 != node) {
        return node->GetParent();
    }
#endif
    return 0;
}
void Layouter::Layout(const AbstractNodePointer& node)
{
…
        Vector2 nodeSize(Layouter::GetSize(*node.ToCanderaObject()));
…
}
```

Touch Support

Touch events are derived from generic events and are equally easy to implement.
For an example of use have a look at the **ToggleButtonWidget** and it's base class **CgiEventWidget**.

- Set EventMask. The EventMask determines which event types will be sent.

```
        const UInt cTbTouchEventExit = CgiEventWidget::TouchEventExit;
        const UInt cTbTouchEventUp = CgiEventWidget::TouchEventUp;
        const UInt cTbTouchEventDown = CgiEventWidget::TouchEventDown;


        Base::SetEventMask(cTbTouchEventExit | cTbTouchEventUp | cTbTouchEventDown);
```

- Override listener function.

```
        protected:

            virtual void OnTouchEvent(TouchEvent touchEvent, Candera::Node
* node, const CgiApplication::MouseEvent& event);
```

- Define your actions inside the listener function **OnTouchEvent**.

```
        switch(touchEvent) {
            case TouchEventDown:
                m_focus = true;
                SetOn(!IsOn());
                break;
            case TouchEventUp:
                if (m_focus) {
                    SendUiEvent(TouchEventClick, 0);
                }
                break;
            case TouchEventExit:
                m_focus = false;
                m_down = !m_down;
                SwitchTexture();
                break;
            default:
                break;
        }
```

Loading of Shaped / Compressed Assets

Loading Shaped Assets

Shaped assets are the result of partitioning an asset into several smaller ones, each containing a part of the original asset, based on a predefined set of rules. Please refer to Shape Asset Library chapter for information on how to create them.

The routine for loading shaped assets is similar to the one presented in Loading an Asset section. In this case, Candera::AssetConfig might be used because it allows the definition of multiple instances of Candera::AssetRepository.

The application should derive from the abstract Candera::AssetConfig in order to support specific Candera::AssetRepository instances. The shaped asset repositories need be added, one by one, to Candera::AssetConfig before initializing Candera::AssetProvider.

Loading shaped assets during runtime

There is also the possibility of adding or removing asset repositories even after the initialization of the asset provider. In this case, the application should derive from Candera::DynamicAssetConfig, which allows the asset configurations to be changed during runtime.

To add a repository during runtime, please use Candera::DynamicAssetConfig::AddRepository. In order to remove one, please use Candera::DynamicAssetConfig::RemoveRepository.

Candera::AssetConfigListener::OnAssetRepositoryAdded should be called in the derived classes of Candera::AssetConfig after a new asset has been added, to notify all listeners and load the asset repository. Call Candera::AssetConfigListener::OnAssetRepositoryRemoved when removing an asset repository.

> When dynamically adding of partitioned assets, the newly added assets should belong to the same original asset that the previous loaded are belonging to.

Using Asset Decompression

Candera AssetLoader is able to support and manage asset data compressed by CGI Studio Asset Shaper Tool. Please refer to Compression section for more information related to data compression.

Using of compressed assets inside the application is be possible only if *CANDERA_ASSET_DECOMPRESSION_ENABLED* variable is set to *true* during the build process of Candera. After enabling this feature, *CANDERA_ASSET_DECOMPRESSION_CHUNK_SIZE* will be enabled for specifying the chunk buffer size for decompression.

A *chunk* represents the buffer size that is used for pulling data from the asset to perform the decompression operation, one chunk at a time. Using a smaller buffer size would be time consuming, so a larger buffer size would be recommended in this case, if enough memory is available.

> *CANDERA_ASSET_DECOMPRESSION_CHUNK_SIZE* is relevant only for Candera::FileAssetRepository.