# 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

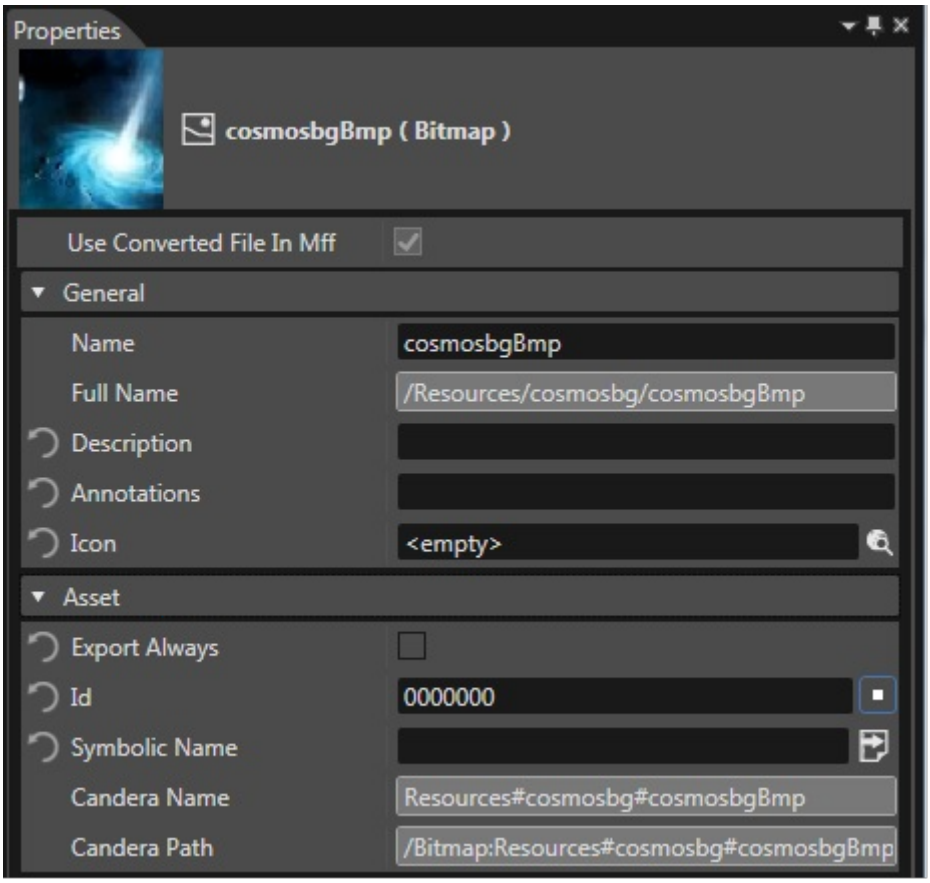
All items in a solution have a type and a name which identifies them in their parent container and a full name which uniquely identifies them in the entire solution. For example, a mesh in a scene could have the full name `/Global/Scenes/MyScene/Group:RootNode/Mesh:MyMesh`.

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
```

---

Revision #11
Created 2023-02-20 06:52:04 UTC by Tsuyoshi.Kato
Updated 2025-01-17 09:21:57 UTC by Elisabeth Zauner