ANSEL INTEGRATION GUIDE

1. SETTING UP

1.1 CODE, COMPILER, AND LINKER CONFIGURATION

The include folder in the Ansel SDK contains all the public header files you will need. A top-level include file, `AnselSDK.h`, is provided for convenience so that you only need to include that one file in your project's code. If you add the path to the Ansel SDK include folder to your project settings this is all you need to gain access to the functionality of the SDK:

```
#include <AnselSDK.h>
```

Note that we do not offer Ansel SDK for static linking: this is intentional. The lib folder in the Ansel SDK contains the import libraries for 32-bit and 64-bit architecture DLLs. Based on the targeted platform architecture you must link in the corresponding import library.

If you wish to delay load the Ansel SDK, a sample project and source files are provided in the samples folder. You can either link in the resulting static lib that this sample produces or simply include the source files directly in your game. Regardless of which route you choose for delay loading you need to define `ANSEL_SDK_DELAYLOAD` for the game and call the `loadAnselSDKLibrary` function that is implemented by the sample. The Ansel SDK needs to be loaded via this function before any call is made to Ansel SDK functions (except the `ansel::isAnselAvailable` function which can always be called - this function is covered in detail later in this document). All other aspects of the Ansel SDK integration should remain the same and no special workflow is required once the Ansel SDK has been loaded.

The redist folder in the Ansel SDK contains the DLLs for 32-bit and 64-bit architecture. These DLLs must be redistributed with the game - if the game is only offered for 64-bit architecture then only 64-bit DLLs should be redistributed (and similarly for 32-bit).

1.2 MACHINE CONFIGURATION

In order to use Ansel you need:

- Windows PC with Windows 7 (32-bit or 64-bit) or newer
- GeForce GTX 600 series or newer
- Ansel-ready display driver. In order to take advantage of all features of a particular version of the SDK a minimum driver version is required. Please consult the README.md included with the SDK for details on this.
- A game using DX11 that has integrated the Ansel SDK

> **NOTE:**
>
> - Support for DX12 is coming soon and options for OpenGL are being investigated.
> - Support for Optimus laptops is coming soon.
> - We currently do not support Ansel for the following NVIDIA GPU / Display configurations
> - Surround
>
> In previous versions of the SDK a whitelisted GeForce game profile was required to activate Ansel for a particular game. This is no longer required - the integration of SDK 1.3 (or later) with a game is enough to enable Ansel with that game (a corresponding minimum driver version is of course also needed as outlined above).

1.3 GAME ENGINE CONFIGURATION

Ansel currently supports the following backbuffer formats:

```
DXGI_FORMAT_R8G8B8A8_UNORM
DXGI_FORMAT_R8G8B8A8_UNORM_SRGB
DXGI_FORMAT_B8G8R8A8_UNORM
DXGI_FORMAT_B8G8R8A8_UNORM_SRGB
DXGI_FORMAT_R10G10B10A2_UNORM
```

Additionally, multisampling is supported for all the above formats. If your game is using a format that is not on the list Ansel will produce images with zero for every pixel - i.e. black.

2. INTEGRATING SDK WITH YOUR GAME

This Ansel SDK uses three major concepts: Configuration, Session and Camera. The Ansel SDK also provides optional Hints and User controls APIs. We will go through each of them in turn and build up an example of game integration in the process.

> **NOTE:** Please consult the header files in the Ansel SDK for reference style documentation.

2.1 CONFIGURATION

As the first step in detecting whether a host computer can support Ansel (correct driver, Ansel enabled, etc) the following function should be called:

```
ANSEL_SDK_API bool isAnselAvailable();
```

This function can be called at any time (prior to calling any other function of Ansel) but it should be called after the game has created its graphics device. If called prior to device creation it will always return false. The function above is useful as the first step in determining if Ansel should be initialized for this game session. If delay loading of the Ansel SDK is being used then a game could for instance choose to only load the Ansel SDK library if the above function returned true. Please note that initialization of Ansel for this particular game can still fail during the `ansel::setConfiguration` call (see below). This means that a game should only advertise Ansel functionality if both `ansel::isAnselAvailable` returns true and `ansel::setConfiguration` succeeds.

During the initialization phase of the game the Ansel configuration should be specified. This is done via the `ansel::Configuration` object. Please consult the header file, `ansel/Configuration.h`, for detailed documentation on each field and default values for each. This is how configuration is typically performed:

```
#include <AnselSDK.h>
...
// during initialization phase of game:
ansel::Configuration config;
// Configure values that we want different from defaults:
config.translationalSpeedInWorldUnitsPerSecond = 5.0f;
config.right =   { -axis_left.x,   -axis_left.y,   -axis_left.z };
config.up =      {  axis_up.x,      axis_up.y,      axis_up.z };
config.forward = {  axis_forward.x, axis_forward.y, axis_forward.z };

config.fovType = ansel::kVerticalFov;
config.isCameraOffcenteredProjectionSupported = true;
config.isCameraRotationSupported = true;
config.isCameraTranslationSupported = true;
config.isCameraFovSupported = true;

config.gameWindowHandle = hWnd;
config.titleNameUtf8 = u8"Best Game Ever";
```

```
config.startSessionCallback = startAnselSessionCallback;
config.stopSessionCallback = stopAnselSessionCallback;
config.startCaptureCallback = startAnselCaptureCallback;
config.stopCaptureCallback = stopAnselCaptureCallback;

auto status = ansel::setConfiguration(config);
if (status != ansel::kSetConfigurationSuccess)
  // Report error, keep calm, and carry on
```

> **NOTE:** Ansel cannot be activated by the player until the configuration has been successfully set. It is therefore advisable to perform the configuration soon after the AnselSDK DLL has been loaded. Also note that a failed `setConfiguration` call is not catastrophic. It means that Ansel cannot by activated by the player so none of the callbacks or any other parts of the SDK will come into use. The programmer is therefore not burdened with having to wrap Ansel SDK calls for the failure case. It is enough to report it for debugging purposes. The `setConfiguration` call will never fail if the integration has been performed correctly.

Let's go through this particular configuration in order.

Games used different units to represent size - it is therefore not possible in Ansel to have a uniform translational speed that works for all games. This is why you will have to set the `translationalSpeedInWorldUnitsPerSecond` to a value that feels right for your game. Move the camera around once Ansel mode has been activated to test your settings; test it also with the accelerator key pressed (see Chapter 3 for Ansel controls)

Note that even though a game must specify how many meters (or fraction of a meter) are in a world unit (see `Configuration::metersInWorldUnit`) this by itself is not enough to derive a default speed that works well for all games. Some games have large worlds where travel is often performed in vehicles (or on mounts) while other games use much smaller worlds and thus much lower travel speeds.

Games use different orientations and chirality (handed-ness) for their coordinate system and camera. The conversion between game and Ansel's internal coordinate system and camera is handled by Ansel. This greatly simplifies integration for game developers. You must specify the unit vectors for right, up, and forward directions that your default oriented game camera uses as part of the configuration step. Once that is done all orientations exchanged between Ansel SDK and the game will be in the game's coordinate system (see section 2.3 for more details). The right, up, and forward directions that you provide must coincide with the default orientation of the game's camera. That is, if no rotation was applied to the camera these are the values of its right, up, and forward axes.

Games will either use vertical or horizontal angle to specify the field of view. The default value in the Ansel configuration is horizontal angle but if your game uses vertical angle you must specify `ansel::kVerticalFov` as the fovType. This will free you from having to convert between the game's field of view and Ansel's internal field of view (see section 2.3 for details).

Not all features of Ansel will necessarily be supported by a game integration. There are also cases where some features of Ansel will be disabled under a particular game scenario. Here we will only discuss the general settings that reflect the Ansel integration with the particular game engine; we will cover specific game scenarios in section 2.2.

In order to support high resolution capture, where screenshots are taken at resolutions higher than display resolution, the game engine needs to support off center projection. In the `Configuration` object you specify if this feature is supported by this particular game integration via the `isCameraOffcenteredProjectionSupported` field. Similarly, support for allowing players to move, orient and zoom the camera is specified via the other supported fields. We will discuss in detail how support for these features is implemented in section 2.3.

The game must provide the window handle to its main window (the window that receives focus when the game is active - this is the same window that receives mouse events). Ansel will automatically hide the Windows mouse cursor when Ansel UI is enabled. Restoring the visibility of the Windows cursor cannot be done reliably by Ansel for all games. It is therefore necessary to call Windows API `ShowCursor` if the game wants the mouse cursor to

be visible after the Ansel UI is closed. For completeness a game can call `ShowCursor(false)` to hide the mouse cursor in the start session callback (this is not necessary) and `ShowCursor(true)` to restore it in the stop session callback (this may be necessary for your game - of course depending on whether the cursor should be visible for the current game state).

Finally we have the session and capture callbacks. The capture callbacks are optional. Those should only be configured if the game uses non-uniform screen based effects that cause issues in multipart shots. An example of an effect that causes issues is vignette. Applying this effect to all the individual tiles of a highres capture or 360 capture will produce poor results. The `startCaptureCallback` will be called when the multipart capture sequence starts and can therefore be used to disable vignette. The `startCaptureCallback` passes a const reference to a `CaptureConfiguration` object containing information about the capture about to be taken. The vignette can then be restored when `stopCaptureCallback` is called. A game integration could of course chose to disable vignette when Ansel session is started but we recommend against this since it would remove the vignette from regular screen shots (non-multipart shot). Note that all these callbacks receive the value of the `userPointer` that is specified in the `Configuration` object.

Usually the version field in the `ansel::Configuration` structure is default initialized (if an object was created using the default constructor and not aggregate initialization or copy from other object for instance) and doesn't need to be set explicitly. In case an object is not created using the default constructor, the version field needs to be set to `ANSEL_SDK_VERSION` defined in `ansel/Version.h`.

The session callbacks are mandatory, they are called when a player wants to begin an Ansel session or end a session. Without these callbacks Ansel cannot be activated. We will discuss them in detail in the next section.

> **NOTE:** All callbacks in Ansel will be triggered from calling D3D Present. They will therefore also happen on the same thread that calls D3D Present.

The `ansel::setConfiguration` function returns its status. The possible return values are:

```
enum SetConfigurationStatus
{
    // successfully initialized the Ansel SDK.
    kSetConfigurationSuccess,
    // the version provided in the Configuration structure is
    // not the same as the one stored inside the SDK
    // binary (header/binary mismatch).
    kSetConfigurationIncompatibleVersion,
    // the Configuration structure supplied for the setConfiguration call is
    // not consistent
    kSetConfigurationIncorrectConfiguration,
    // the Ansel SDK is delay loaded and setConfiguration is called before
    // the SDK is actually loaded
    kSetConfigurationSdkNotLoaded
};
```

The integration should only continue as normal (with regard to the Ansel functionality at least) when `kSetConfigurationSuccess` is returned. In case `kSetConfigurationIncompatibleVersion` is returned, most likely the Ansel SDK binary was obtained without updating the headers, which also contain a version. Every update of the Ansel SDK requires a new build of a game. In case `kSetConfigurationIncorrectConfiguration` is returned, on of the following fields could be set incorrectly:

- `right`, `up`, `forward` vectors do not form an orthogonal basis
- `startSessionCallback` or `stopSessionCallback` is nullptr
- rotational or translational speed multipliers are zero
- `fovType` is neither horizontal, nor vertical
- `gameWindowHandle` field is not set

In case `kSetConfigurationSdkNotLoaded` is returned, the reason is that the Ansel SDK is delay loaded and `setConfiguration` is called before loading the library via `loadAnselSDKLibrary` (see code under samples folder).

2.2 SESSION

The time period from when a player successfully starts Ansel and until Ansel is stopped is called a session. A session is collaboratively started and operated between the game and Ansel. When a player requests a session start (for example by pressing ALT+F2) Ansel will call the registered session start callback. It is however expected that Ansel cannot always be activated. The game may for instance be on a loading screen or playing a movie sequence. The callback should then immediately return with `Ansel::kDisallowed` return value.

During an Ansel session the game:

- Must stop drawing UI and HUD elements on the screen, including mouse cursor
- Must call `ansel::updateCamera` on every frame (see section 2.3 for details)
- Should pause rendering time (i.e. no movement should be visible in the world)
- Should not act on any input from mouse and keyboard and must not act on any input from gamepads

The function signatures of the session related callbacks are listed below.

```
enum StartSessionStatus
{
    kDisallowed = 0,
    kAllowed
};

typedef StartSessionStatus(*StartSessionCallback)(SessionConfiguration& settings,
                                                  void* userPointer);

typedef void(*StopSessionCallback)(void* userPointer);
```

As you will notice the start callback receives an additional `SessionConfiguration` object. This object must reflect the configuration of the currently activated session if `Ansel::kAllowed` is returned from the callback. Let's take a look at this configuration object:

```
struct SessionConfiguration
{
    // User can move the camera during session
    bool isTranslationAllowed;
    // Camera can be rotated during session
    bool isRotationAllowed;
    // FoV can be modified during session
    bool isFovChangeAllowed;
    // Game is paused during capture
    bool isPauseAllowed;
    // Game allows highres capture during session
    bool isHighresAllowed;
    // Game allows 360 capture during session
    bool is360MonoAllowed;
    // Game allows 360 stereo capture during session
    bool is360StereoAllowed;
    // The maximum FoV value in degrees displayed in the Ansel UI.
    // Any value in the range [140, 179] can be specified
    // and values outside will be clamped to this range.
    float maximumFovInDegrees;
    // Default constructor not included here
};
```

As you can see each session has fine grained control over what features of Ansel are offered to the player. This is to support different contexts that the game may be in - where some features of Ansel may not be desired. For instance, let's say that a game uses in-engine movie sequences. Ansel could certainly be used to take regular and highres screenshots during those sequences. However, the game developers may wish to prohibit any player controlled camera movement or 360 captures during those sequences since they could expose geometry that was never built because the sequences have been carefully orchestrated. This is what such a callback could look like:

```
ansel::StartSessionStatus startAnselSessionCallback(ansel::SessionConfiguration& conf,
                                                    void* userPointer)
{
    if (isGameLoading || isGameInMenuScreens)
        return ansel::kDisallowed;

    if (isGameCutScenePlaying)
    {
        conf.isTranslationAllowed = false;
        conf.isRotationAllowed = false;
        conf.isFovChangeAllowed = false;
        conf.is360MonoAllowed = false;
        conf.is360StereoAllowed = false;
    }

     g_isAnselSessionActive = true;

      return ansel::kAllowed;
}
```

> **NOTE:** The final feature set presented in the Ansel UI is always a combination of the global configuration specified via the `ansel::Configuration` object and the particular session configuration specified via the `ansel::SessionConfiguration` object. For instance, if off-center projection is not supported by the integration and this is marked as such in the global configuration then the `isHighresAllowed` setting will have no effect for an Ansel session because the feature is simply not supported by the integration.

> Similarly, if `isRotationAllowed` is set to false for the session then no form of 360 capture will be possible and hence the `is360MonoAllowed` and `is360StereoAllowed` will have no effect. In the sample code above we still set them but this is done for clarity and completeness.

In a session where rendering time cannot be paused this can be communicated with the isPauseAllowed setting. This could for instance be the case in a game that offers multiplayer game modes. That being said, some multiplayer game engines still allow rendering time to be frozen - this just means that the state of the world will have advanced when the Ansel session ends. The stitcher developed for Ansel does not currently support feature detection or other methods used to handle temporal inconsistencies. This means that multipart shots (highres and 360) are not supported during sessions when pause is disallowed.

> **NOTE:** A hybrid approach may be desirable during a multiplayer game mode. In this scenario the `isPauseAllowed` setting would be
> set to true during the start of the session but the game would still be updating during the Ansel session. This would allow players to
> see what is going on in the game and end the Ansel session if circumstances require it (getting attacked, etc). When a multipart shot
> is taken (where the rendering time needs to be frozen) Ansel will always trigger the `startCaptureCallback` and the game can use this

> callback to freeze rendering time *only during the capture of the multipart shots* since `stopCaptureCallback` will be triggered at the
> end of the capture sequence. This would result in getting correct highres and 360 captures while allowing the player to be aware of how
> the game is evolving during the Ansel session.

The stop session callback is called when the player requests to end the session (for instance by pressing ALT+F2). This function is only called if the previous call to start session returned `ansel::kAllowed`. The matching function to what we implemented above would look like this:

```
void stopAnselSessionCallback(void* userPointer)
{
    g_isAnselSessionActive = false;
}
```

A game can trigger a session start or stop (which leads to the above mentioned callbacks being eventually called). This is done using the following interface from `Session.h`:

```
ANSEL_SDK_API void startSession();
ANSEL_SDK_API void stopSession();
```

This can be used to enable activation of Ansel UI via gamepad for instance. A game can use its existing gamepad handling to call `ansel::startSession()` if for instance left-stick is pressed (and a session is not already active). Similarly, `ansel::stopSession()` should be called when the same action is performed and session is active. The game is expected to track whether a session is active via the start/stop session callbacks that were outlined earlier in this section. Other means of Ansel activation could of course be employed (different keyboard shortcut, etc) and the game could call `startSession` / `stopSession` accordingly.

2.3 CAMERA

The camera object acts as the communication channel between Ansel and the game. The concepts described earlier are either used for one-off configuration or rare events. Once an Ansel session has been started the game needs to call Ansel on every frame to update the camera. It's helpful to first look at the `Camera` interface:

```
struct Camera
{
    nv::Vec3 position;
    nv::Quat rotation;
    float fov;
    float projectionOffsetX, projectionOffsetY;
};

ANSEL_SDK_API void updateCamera(Camera& camera);
```

As noted before the header file contains documentation for each field. Here it suffices to say that the values are all in the game's coordinate system, since this has been established via the `ansel::setConfiguration` call (see section 2.1). The field of view, fov, is in degrees and in the format specified during the previously mentioned call. The final two values are used to specify the off-center projection amount. Let's illustrate how this all works with sample code following the earlier session callbacks we had created:

```
if (g_isAnselSessionActive)
{
  if (!was_ansel_in_control)
  {
    store_original_camera_settings();
    was_ansel_in_control = true;
```

```
  }
  ansel::Camera cam;
  cam.fov = get_game_fov_vertical_degrees();
  cam.position = { game_cam_position.x, game_cam_position.y,
                   game_cam_position.z };
  cam.rotation = { game_cam_orientation.x,
                   game_cam_orientation.y,
                   game_cam_orientation.z,
                   game_cam_orientation.w };

  ansel::updateCamera(cam);
  // This is where a game would typically perform collision detection
  // and adjust the values requested by player in cam.position

  game_cam_position.x = cam.position.x;
  game_cam_position.y = cam.position.y;
  game_cam_position.z = cam.position.z;

  game_cam_orientation.x = cam.rotation.x;
  game_cam_orientation.y = cam.rotation.y;
  game_cam_orientation.z = cam.rotation.z;
  game_cam_orientation.w = cam.rotation.w;

  set_game_fov_vertical_degrees(cam.fov);

  // modify projection matrices by the offset amounts -
  // we will explore this in detail separately
  offset_game_projection_matrices(cam.projectionOffsetX, cam.projectionOffsetY);

  return;
}
else
{
  if (was_ansel_in_control)
  {
    restore_original_camera_settings();
    was_ansel_in_control = false;
  }
}
```

The sample above is a full implementation of Ansel support, in the sense that it supports camera translation, camera rotation, changing of field of view, and offset projection. Most rendering engines employ a view matrix; the camera is most often just a game side concept. Fortunately the view matrix is just the inverse of the camera matrix so it's easy to obtain. Remember that the camera matrix is established by transforming the default camera by the camera rotation. Since some code bases use matrices rather than quaternions for rotations we provide a couple of utility functions in the Camera interface:

```
ANSEL_SDK_API void quaternionToRotationMatrixVectors(const nv::Quat& q,
                                                     nv::Vec3& right,
                                                     nv::Vec3& up,
                                                     nv::Vec3& forward);

ANSEL_SDK_API void rotationMatrixVectorsToQuaternion(const nv::Vec3& right,
                                                     const nv::Vec3& up,
                                                     const nv::Vec3& forward,
                                                     nv::Quat& q);
```

The former one can be used to convert the `ansel::Camera::rotation` to a matrix that can then be used to transform the basis vectors of the game's default oriented camera to establish the camera matrix. The latter can be used to convert the game's camera rotation matrix to a quaternion that can be passed into `ansel::Camera::rotation` prior to calling `ansel::updateCamera`.

Even though the sample above is a full implementation of Ansel support it does however not employ any collision detection or other constraints on the camera movement. This is unrealistic since most games will at least want to limit the range the camera can travel. These limitations will always be specific to the game in question and collision is best handled by the game using the systems that are already in place. We do therefore not elaborate on this piece here. It should still be noted that Ansel is stateless when it comes to position so the game can adjust the position (based on collision response or constraints) any way it sees fit. The new position will always be communicated to Ansel on the next frame, via `ansel::updateCamera`.

Another aspect that is not covered in the sample code above is the handling of projection offset. We will explore that aspect in more detail here.

The `projectionOffsetX` and `projectionOffsetY` members of the `ansel::Camera` specify the amount that the projection matrix needs to be offset by. These values are normalized coordinates applied directly to the projection matrix, they should not need any scaling. These values are only used by Super Resolution captures - for all other capture types they will be zero. Expanding on the sample code above:

```
void offset_game_projection_matrices(float offsetX, float offsetY)
{
  // In this simple example we only need to modify the projection
  // matrix associated with the game camera. If the game is doing
  // clever things like optimizing reflections or shadows based on
  // projection matrix then those code paths need to take a non-zero
  // projection offset into account.

  // For nostalgia effect this game is using an old classic:
  D3DXMATRIX projection;
  D3DXMatrixPerspectiveFovRH(&projection, g_fov_radians, g_aspect, g_z_near, g_z_far)

  // Apply the offsets directly to the finished product (values are already normalized):
  projection._31 += offsetX;
  projection._32 += offsetY;

  // Update the games projection matrix:
  g_projection_matrix = projection;
}
```

2.4 HINTS

To perform Raw capture (to EXR format) Ansel tries to use certain heuristics to detect which of the game buffers contains HDR pixel data. These heuristics won't work for all games. For instance, they won't work for games that use deferred contexts with multithreaded rendering. For those scenarios the hinting API can be used (see `ansel/Hints.h`). Besides HDR color buffer, Ansel also can also use depth or HUDless buffers to apply effects which needs these types of buffers (e.g. depth of field).

Below we will outline how the hinting API is used to make capture work under these circumstances. If game developers do not want to use the hinting API to make raw capture work the `isRawAllowed` setting in the `ansel::SessionConfiguration` should be set to false during the startSession callback. This will disable the 'Raw' option in the Ansel UI.

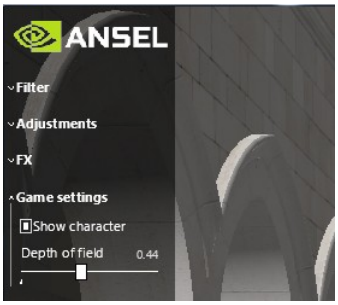
```
  // Call this right before setting HDR render target active
  // bufferType is an optional argument specifying what type of buffer is this -
  // an HDR color buffer, a depth buffer or HUDless buffer. The default option is HDR color buffer.
  // hintType is an optional argument specifying what type of hint is this -
```

```
// it could be called after or before the bind of a buffer that this hint marks.
// The default option is kHintTypePreBind, which means the hint should be called before
// the render target is bound.
// threadId is an optional argument allowing Ansel to match the thread which calls
// SetRenderTarget (or analogous function, since this is graphics API dependent)
// to the thread which called the hint. The default value of kNoMatching
// means that no such matching is going to happen. The special value of 0 means that
// Ansel SDK is going to match thread ids automatically. Any other value means a specific thread id
// known at integration side.
ANSEL_SDK_API void markBufferBind(BufferType bufferType = kBufferTypeHDR,
                                  HintType hintType = kHintTypePreBind,
                                  uint64_t threadId = kThreadingBehaviourNoMatching);
// Call this right after the last draw call into the HDR render target
// bufferType is an optional argument specifying what type of buffer is this -
// an HDR color buffer, a depth buffer or HUDless buffer. The default option is HDR color buffer.
// threadId is an optional argument allowing Ansel to match the thread which calls
// SetRenderTarget (or analogous function, since this is graphics API dependent)
// to the thread which called the hint. The default value of kNoMatching
// means that no such matching is going to happen. The special value of 0 means that
// Ansel SDK is going to match thread ids automatically. Any other value means a specific thread id
// known at integration side.
ANSEL_SDK_API void markBufferFinished(BufferType bufferType = kBufferTypeHDR,
                                      uint64_t threadId = kThreadingBehaviourNoMatching);
```

To identify the HDR buffer for Ansel call `markBufferBind` before binding the buffer to the graphics pipeline. In case the buffer contents is not overwritten before calling `Present` (or `glSwapBuffers()`) it is fine to not call `markBufferFinished`. In case the same buffer is reused for other purposes and at the time the `Present` (or `glSwapBuffers`) gets called its content does not represent the framebuffer calling `markBufferFinished` at the moment where the buffer is not used anymore but before it is unbound from the graphics pipeline is necessary. Both functions have threadId argument, which is an optional argument allowing Ansel to match the thread calling a hint and a particular graphics API call that some other game thread might perform. The default value of 0xFFFFFFFFFFFFFFFFull means that no such matching should happen. The special value of 0 means that Ansel SDK is going to match thread ids automatically. Any other value means a specific thread id known at integration side. `markBufferBind` can be called before or after binding a buffer to the graphics pipeline if appropriate `hintType` is selected. Use `bufferType` other than kBufferTypeHDR to mark other types of buffers. It is advisable to mark all types of buffers supported by Ansel SDK, otherwise some of Ansel functions wont work.

2.5 USER CONTROLS

Sometimes a game developer is willing to expose some of the game properties in the Ansel UI. One example of this could be a checkbox allowing to hide and unhide the main character. Another example would be a slider allowing changing the amount of in-game depth of field effect or any other effect. To achieve that the Ansel SDK provides an optional User controls API.



The basics of the API look like this:

```
// This function adds a user control defined with the UserControlDesc object
ANSEL_SDK_API UserControlStatus addUserControl(const UserControlDesc& desc);
// Specifies a translation for a control label.
// This function requires a valid lang (a-la "en-US", "es-ES", etc) and a label
// (non nullptr and non empty string without '\n', '\r' and '\t') encoded in utf8.
// The length of the label should not exceed 20 characters not counting the
// trailing zero
ANSEL_SDK_API UserControlStatus setUserControlLabelLocalization(uint32_t userControlId,
                                                                const char* lang,
                                                                const char* labelUtf8);
// This function removes a control that was added previously
ANSEL_SDK_API UserControlStatus removeUserControl(uint32_t userControlId);
// This function returns the current control value
ANSEL_SDK_API UserControlStatus getUserControlValue(uint32_t userControlId, void* value);
// This function sets the current control value
ANSEL_SDK_API UserControlStatus setUserControlValue(uint32_t userControlId, void* value);
```

Notice, it is allowed to call any of these functions at any time, even before `ansel::setConfiguration`. Currently the supported types of controls are a slider and a checkbox. In order to create a control it is required to create an object of `ansel::UserControlDesc`, set it up appropriately and call the `ansel::addUserControl` function. The function also requires a callback of type `UserControlCallback` to be passed. This callback will be called every time user changes the control value in the UI. Please consult the documentation in the headers to learn how to setup the `ansel::UserControlDesc` structure. It is important to notice that `UserControlInfo::value` is used both for adding a control and receiving its value. In either case this pointer should never be nullptr. Its lifetime is either limited to the `UserControlCallback` execution or should be longer than `ansel::addUserControl` execution.

Finally, here is an example of how this API could be used to add a checkbox with additional label localizations:

```
ansel::UserControlDesc characterCheckbox;
characterCheckbox.labelUtf8 = "Show character";
characterCheckbox.info.userControlId = 0;
characterCheckbox.info.userControlType = ansel::kUserControlBoolean;
const bool defaultValue = true;
characterCheckbox.info.value = &defaultValue;
characterCheckbox.callback = [](const ansel::UserControlInfo& info) {
  g_renderCustomUI = *reinterpret_cast<const bool*>(info.value);
};
ansel::addUserControl(characterCheckbox);
ansel::setUserControlLabelLocalization(characterCheckbox.info.userControlId,
                                       "ru-RU", russianTranslationUtf8);
ansel::setUserControlLabelLocalization(characterCheckbox.info.userControlId,
                                       "es-ES", spanishTranslationUtf8);
...
```

In order to setup a control with the current value (e.g. time of day) it is recommended to use `getUserControlValue` / `setUserCont` functions inside startAnselSessionCallback. Here is an example:

```
// Ansel SDK initialization

// choose any value for the user control id
const uint32_t timeOfDayUCID = 42u;
ansel::UserControlDesc timeOfDaySlider;
characterCheckbox.labelUtf8 = "Time of day";
characterCheckbox.info.userControlId = timeOfDayUCID;
characterCheckbox.info.userControlType = ansel::kUserControlSlider;
const float defaultValue = 0.5f;
characterCheckbox.info.value = &defaultValue;
characterCheckbox.callback = [](const ansel::UserControlInfo& info) {
```

```
      g_timeOfDay = *reinterpret_cast<const float*>(info.value);
    };
    ansel::addUserControl(timeOfDaySlider);
    ...
}

ansel::StartSessionStatus startAnselSessionCallback(
       ansel::SessionConfiguration& conf, void* userPointer)
{
    ...
    // initialize Ansel UI with the latest true time of day value
    ansel::setUserControlValue(timeOfDayUCID, &g_timeOfDay);
    ....
}
```

3. TAKING PICTURES WITH ANSEL

3.1 ACTIVATING AND DEACTIVATING ANSEL

Players can start/stop Ansel session by pressing ALT+F2. Other ways of activation are possible if the game has used the `ansel::startSession` and `ansel::stopSession` functions in the Ansel SDK to implement other triggers (see section 2.2)

3.2 MOVING THE CAMERA

The camera can be moved via keys WASD and XZ for up/down. The camera can also be moved with the left stick on a gamepad and trigger buttons for up/down. Movement can be accelerated by holding SHIFT on keyboard or depressing right stick on gamepad.

3.3 ROTATING THE CAMERA

The yaw and pitch of the camera is directly controlled by mouse or right stick on gamepad. The roll of the camera is adjusted via the user interface Roll slider but can also be controlled directly via shoulder buttons on gamepad.

3.4 APPLYING A FILTER

A number of filters can be selected via the Filter slider. Some filters, like 'Custom' have additional settings that can be used to adjust the filter even further.

3.5 TAKING A PICTURE

Ansel offers the following capture types (selected via the Capture type slider):

- Screenshot
- Highres
- 360
- Stereo
- 360 Stereo

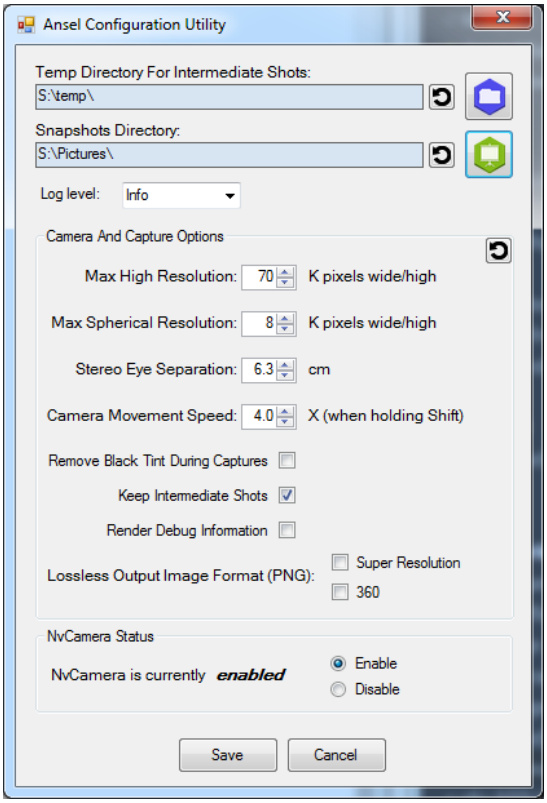
Not all of these capture types may be available since it depends on the game integration and the current session (see sections 2.1 and 2.2). Once a type has been chosen the picture is taken by pressing Snap button. Some pictures may take significant time to produce, especially highres shots of large dimensions. If the game uses streaming the streaming performance may be affected when shots involving many parts are being stitched together.

> **NOTE:** Not all filters are valid with multipart Capture types (360 and Highres). You may therefore see filters (or aspects of a filter) removed in the final picture.

4. TROUBLESHOOTING AND DEBUGGING COMMON PROBLEMS

In this section we collected commonly occurring problems we've seen while integrating Ansel with games. This section can hopefully help you resolve a problem or two. It is generally useful to be able to inspect the individual shot tiles that are captured when generating pictures that require multiple shots. Locate the `NvCameraConfiguration.exe` utility. It can be found inside

Run the utility. A screen similar to this one should appear:



Check the 'Keep Intermediate Shots' option so that you can inspect the individual tiles. You can also pick a different location to store the tiles by changing the 'Temp Directory for Intermediate Shots'.

Notice the 'Log level' setting. By default this is set to 'Disabled', which means that no logs are produced. If you set this to for instance 'Info' then all log messages with severity Info and higher will be emitted to a log file. Log files are stored in the current users profile directory, inside an 'ansel' folder. The path is `%USERPROFILE%\ansel`. Each log file is named after the associated executable and includes a timestamp. Please note that multiple log files can be produced by the same game during one session since each device starts a separate Ansel logging session. That being said only one device (and therefore log file) will be associated with the presentable surface and will thus contain the most helpful information.

4.1 ANSEL CANNOT BE ACTIVATED

There are a number of configuration issues that can result in Ansel not activating for your game when you press the activation keys (ALT+F2). The following list is helpful in debugging the issue:

1. Verify that Ansel is enabled for the machine. You can do this by running the `NvCameraEnable.exe` utility with no arguments (this tool was introduced in section 1.2). It will return `1` if Ansel is enabled, `0` otherwise. You can change this setting by issuing `NvCameraEnable.exe on` or using the GUI tool `NvCameraConfiguration.exe` that was introduced in the beginning of this chapter.
2. If the game executable is not whitelisted for Ansel (by the installed driver) you can disable whitelisting checks as outlined in section 1.2.
3. Verify that the hardware and software requirements from sections 1.2 and 1.3 are met by the machine and game.
4. If you are running on a Hybrid/Optimus computer that has both a integrated GPU and a discreet NVIDIA GPU, make sure the game is running on the NVIDIA GPU. Use the NVIDIA Control Panel to verify/configure this setting for the game.
5. Verify that Ansel is trying to start by setting a breakpoint on the `startSessionCallback`.
6. Depending on which phase is blocking Ansel activation, logging can be helpful. See the beginning of this chapter on how to utilize logging.

4.2 ARTEFACTS IN MULTIPART SHOTS

This is where we cover the most common errors we've seen while capturing multipart shots in games.

4.2.1 Image tiles suffer from "acne"

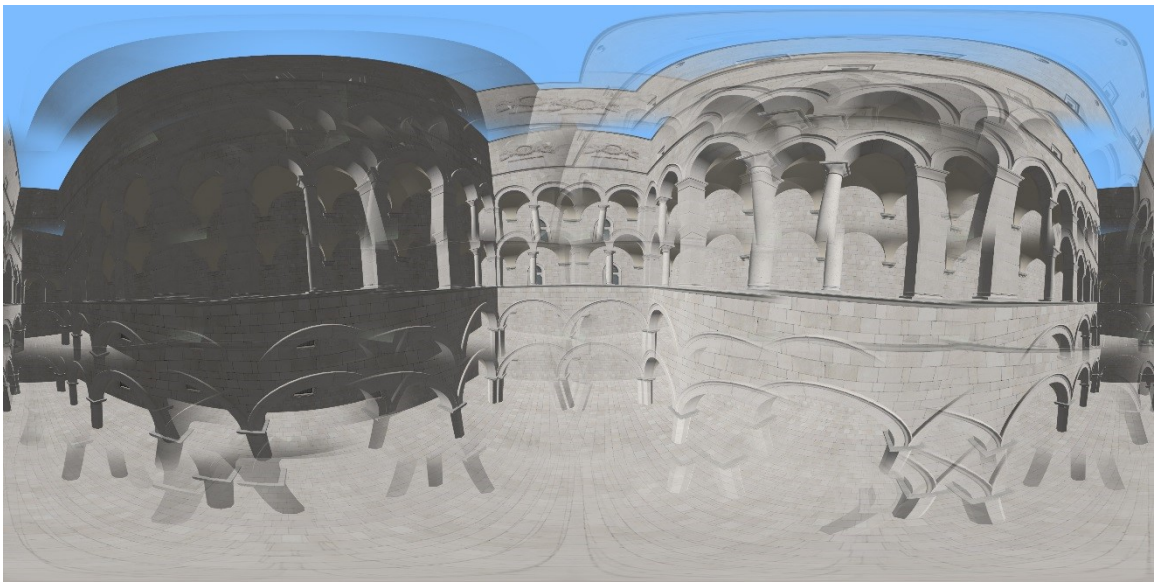
This is probably best described with images. Here is a tile exhibiting the problem:

Note the "acne" to the left of the tree, caused by temporal anti-aliasing. This can be fixed in two different ways. One method is to use the `captureSettleLatency` field in the configuration object. This field specifies how many frames Ansel should wait when taking multipart shots for the frame to settle - i.e. for any temporal effects to have settled. In this specific case (where the image is taken from) it takes one frame so the value should be 1. This is what the image looks like with that setting:

Another way to solve this problem is to harness the `startCapture` / `stopCapture` callbacks to disable temporal AA. This effectively disables the temporal AA feature during the multipart capture sequence.
Which solution should you use? Well, it depends. You need to subjectively evaluate how much the temporal AA enhances image quality vs the cost of waiting for the frame to settle. At a `captureSettleLatency` of 1 the cost is rather small and thus weighs in favor of using that solution.

4.2.2 Ghosting everywhere in final picture

Usually it looks something like this:



Most often this is the result of incorrect field of view being submitted to Ansel - or error made on conversion or usage of value coming back from Ansel. It is recommended that you match the field of view type between game and Ansel to avoid any conversion mistakes. See section 2.1 on how you can configure Ansel to use the game's field of view.

4.2.3 Screen space reflections fade out with increased Highres capture resolution

Below is a regular screenshot taken with Ansel:

If we now select capture type Highres with a large enough multiplier we get this picture (scaled down in resolution to fit this document):

There is unfortunately no workaround for this problem, it is a limitation of the capture method used.

4.2.4 It's all a blur

Motion blur needs to be disabled during multipart capture. Otherwise results like this can be produced when taking 360 picture:

4.2.5 Streaky reflections

When enabling highres capture for your game you may witness results similar to this:

The reason may be that the projection offset and reduced field of view employed by the highres capture method is not being accounted for in the game's reflection code path.

4.3 THE VIEW OF THE WORLD "POPS" WHEN ENTERING AND EXITING ANSEL MODE

This is typically due to incorrect field of view being passed on the first frame or due to a screen space effect being disabled when Ansel mode is activated. For the latter it is preferred to deactivate troublesome effects only during multipart captures (via the capture callback).

4.4 GAME MOUSE CURSOR IS NOT VISIBLE AFTER RETURNING TO GAME FROM ANSEL

The visible state of the mouse cursor needs to be restored by calling Windows API `SetCursor(true)` when the Ansel session has ended (after `ansel::Configuration::stopSessionCallback` is triggered).

4.5 DOUBLE MOUSE CURSORS

Strictly speaking UI and HUD elements must not be rendered when game is in Ansel mode and this includes any cursors. We have however experienced the situation where the game renders a mouse cursor on top of its window when it regains focus. If the game is in Ansel mode this will result in two moving mouse cursors - a very confusing experience indeed. As mentioned the game shouldn't be rendering a mouse cursor when Ansel is active. That being said we have a mechanism to prevent this from happening - this mechanism is enabled if the game passes its window handle during configuration, in the `Configuration::gameWindowHandle` field.

4.6 CAMERA ROTATION OR MOVEMENT IS INCORRECT

Incorrect rotation is best observed with a gamepad - i.e. pushing the joystick left doesn't rotate the view towards the left or pushing the joystick up doesn't rotate the view up. Incorrect movement can be verified with either keyboard or gamepad.

This problem is usually rooted in incorrect axes provided for right, up, and down directions in the `ansel::Configuration` struct. See section 2.1.

4.7 CAMERA ROTATION OR MOVEMENT IS TOO SLOW / TOO FAST

The speed for rotation is set via the `rotationalSpeedInDegreesPerSecond` field during configuration. The default value is 45 degrees/second. The speed for movement is set via the `translationalSpeedInWorldUnitsPerSecond` field during configuration. The default value is 1 world unit/second.

4.8 ALL ANSEL CAPTURES PRODUCE BLACK IMAGES

This is usually caused by an unsupported backbuffer format. Please consult section 1.3 for a list of supported formats.

4.9 ALL ANSEL CAPTURES ARE NAMED AFTER THE GAME EXECUTABLE

This happens because the game does not have a GeForce profile (yet) and a setting for `titleNameUtf8` was not specified when `ansel::setConfiguration` was called. It is best to provide a name for the title via `titleNameUtf8` since this will work regardless of whether the driver being used has a GeForce profile for the game.

4.9 RAW CAPTURE FAILS

By default Ansel uses heuristics to try to identify the raw HDR buffer. For some games these heuristics won't work. Section 2.4 contains the information on how to address this.

APPENDIX A

Notice

The information provided in this specification is believed to be accurate and reliable as of the date provided. However, NVIDIA Corporation ("NVIDIA") does not give any representations or warranties, expressed or implied,

as to the accuracy or completeness of such information. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This publication supersedes and replaces all other specifications for the product that may have been previously supplied.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to this specification, at any time and/or to discontinue any product or service without notice. Customer should obtain the latest relevant specification before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer. NVIDIA hereby expressly objects to applying any customer general terms and conditions with regard to the purchase of the NVIDIA product referenced in this specification.

NVIDIA products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this specification. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this specification, or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this specification. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA. Reproduction of information in this specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ROVI Compliance Statement

NVIDIA Products that support Rovi Corporation's Revision 7.1.L1 Anti-Copy Process (ACP) encoding technology can only be sold or distributed to buyers with a valid and existing authorization from ROVI to purchase and incorporate the device into buyer's products.

This device is protected by U.S. patent numbers 6,516,132; 5,583,936; 6,836,549; 7,050,698; and 7,492,896 and other intellectual property rights. The use of ROVI Corporation's copy protection technology in the device must

be authorized by ROVI Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by ROVI Corporation. Reverse engineering or disassembly is prohibited.

OpenCL
OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks
NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

(c) 2016 NVIDIA Corporation. All rights reserved.

www.nvidia.com