Display Driver

Example

Simple display driver example, that passes 8 bit RGBA pixels to a specified callback.

driver_display_callback.cpp

```
#include <ai.h>

namespace ASTR {
   static const AtString callback("callback");
   static const AtString callback_data("callback_data");
   static const AtString color_space("color_space");
};

AI_DRIVER_NODE_EXPORT_METHODS(DriverDisplayCallbackMtd)

typedef void (*DisplayCallback)(uint32_t x, uint32_t y, uint32_t width, uint32_t height, uint8_t* buffer,
void* data);

node_parameters
{
   AiParameterPtr("callback"     , NULL  );
   AiParameterPtr("callback_data", NULL  );  // This value will be passed directly to the callback function
}

node_initialize
{
   AiDriverInitialize(node, false);
}

node_update
{
}

driver_supports_pixel_type
{
   switch (pixel_type)
   {
      case AI_TYPE_FLOAT:
      case AI_TYPE_RGB:
      case AI_TYPE_RGBA:
         return true;
      default:
         return false;
   }
}

driver_extension
{
   return NULL;
}

driver_open
{
}

driver_needs_bucket
{
   return true;
}

driver_prepare_bucket
{
   DisplayCallback cb = (DisplayCallback) AiNodeGetPtr(node, ASTR::callback);
```

```
   // Call the callback function with a NULL buffer pointer, to indicate
   // a bucket is going to start being rendered.
   if (cb)
   {
      void *cb_data = AiNodeGetPtr(node, ASTR::callback_data);
      (*cb)(bucket_xo, bucket_yo, bucket_size_x, bucket_size_y, NULL, cb_data);
   }
}

driver_write_bucket
{
   int pixel_type;
   const void* bucket_data;

   // Get the first AOV layer
   if (!AiOutputIteratorGetNext(iterator, NULL, &pixel_type, &bucket_data))
      return;

   const bool dither = true;

   // Retrieve color manager for conversion
   AtNode* color_manager = (AtNode*)AiNodeGetPtr(AiUniverseGetOptions(), "color_manager");
   AtString display_space, linear_space;
   AiColorManagerGetDefaults(color_manager, display_space, linear_space);
   if (!display_space)
      display_space = linear_space;

   // Allocates memory for the final pixels in the bucket
   //
   // This memory is not released here. The client code is
   // responsible for its release, which must be done using
   // the AiFree() function in the Arnold API
   uint8_t* buffer = (uint8_t*)AiMalloc(bucket_size_x * bucket_size_y * sizeof(uint8_t) * 4);
   int minx = bucket_xo;
   int miny = bucket_yo;
   int maxx = bucket_xo + bucket_size_x - 1;
   int maxy = bucket_yo + bucket_size_y - 1;

   for (int j = miny; (j <= maxy); ++j)
   {
      for (int i = minx; (i <= maxx); ++i)
      {
         int bx = i - minx;
         int by = j - miny;
         AtRGBA source = AI_RGBA_ZERO;

         switch (pixel_type)
         {
            case AI_TYPE_FLOAT:
            {
               float f = ((float*)bucket_data)[by * bucket_size_x + bx];
               source = AtRGBA(f, f, f, 1.0f);
               break;
            }
            case AI_TYPE_RGB:
            {
               AtRGB rgb = ((AtRGB*)bucket_data)[by * bucket_size_x + bx];
               source = AtRGBA(rgb, 1.0f);
               break;
            }
            case AI_TYPE_RGBA:
            {
               source = ((AtRGBA*)bucket_data)[by * bucket_size_x + bx];
               break;
            }
         }

         AiColorManagerTransform(color_manager, display_space, false, false, NULL, (uint8_t*)&source.rgb());
```

```
         uint8_t* target = &buffer[(by * bucket_size_x + bx) * 4];
         target[0] = AiQuantize8bit(i, j, 0, source.r, dither);
         target[1] = AiQuantize8bit(i, j, 1, source.g, dither);
         target[2] = AiQuantize8bit(i, j, 2, source.b, dither);
         target[3] = AiQuantize8bit(i, j, 3, source.a, dither);
      }
   }

   // Sends the buffer with the final pixels to the callback for display.
   //
   // The callback receives ownership over this buffer, so it must
   // release it when it is done with it, using the AiFree() function
   // in the Arnold API.
   //
   // The reason for doing this is to decouple this code from the visualization
   // process, so, as soon as the buffer is ready, this driver will send it to
   // the callback and return to the rendering process, which will continue
   // asynchronously, in parallel with the visualization of the bucket, carried
   // out by the client code.
   //
   DisplayCallback cb = (DisplayCallback) AiNodeGetPtr(node, ASTR::callback);
   if (cb)
   {
      void *cb_data = AiNodeGetPtr(node, ASTR::callback_data);
      (*cb)(bucket_xo, bucket_yo, bucket_size_x, bucket_size_y, buffer, cb_data);
   }
}

driver_process_bucket
{
   // Use this instead of driver_write_bucket for best performance, if your
   // callback handling code is thread safe.
}

driver_close
{
}

node_finish
{
}

node_loader
{
   if (i>0)
      return false;

   node->methods      = DriverDisplayCallbackMtd;
   node->name         = "driver_display_callback";
   node->node_type    = AI_NODE_DRIVER;
   strcpy(node->version, AI_VERSION);
   return true;
}
```