ARStakeout

AR Overlays for VIS with Vulkan

PART OF HEXAGON

- when it has to be **right**

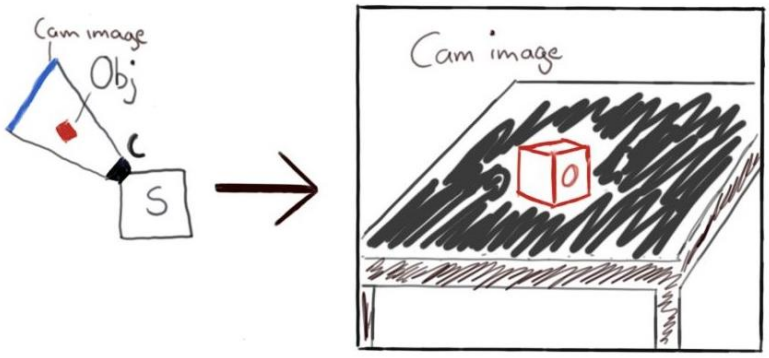
Leica Geosystems

Juni - August 2018
Patrick Gantner (ganp)
patrick.gantner@leica-geosystems.com
Supervisor: Thomas Mörwald (moet)
thomas.moerwald@leica-geosystems.com
Leica Geosystems AG
Heerbrugg, Switzerland

Concept

Using the existing Visual Inertial System (VIS) for pose estimation, BIM (Building Information Management) information is displayed using Augmented Reality technology. Specific use-cases, for example, stakeout or CAD previews are implemented to demonstrate the advantages of such overlays.

The focus of the implementation is on creating a correct representation of the physical camera in the virtual scene and render objects on top of the incoming video stream of the device. For a good AR experience the objects should be distorted according to the camera's distortion model and be occluded correctly with the environment.



1 Schemantic AR operation

A first prototype is implemented on an existing Leica platform with the use of Leica VIS library, Vulkan and OpenCV. Initially, the target platform is a Windows PC running Windows 10 and allowing offline (test data sets) workflows.
The focus of the first prototype is to demonstrate the usability and possible use-cases in combination with current or future Leica devices. The prototype may also set up a basis for workflows using VIS and AR.

2018-23-02
Name: Patrick Gantner
E-Mail: patrick.gantner@leica-geosystems.com

Leica Geosystems AG
Heinrich-Wild-Strasse
CH-9435 Heerbrugg
Switzerland
Phone +41 71 727 31 31
www.leica-geosystems.com

Rendering Engine

The demo application is developed using the latest[1] Vulkan SDK and MSVC140. The creation of windows and input handling is done with the help of GLFW and for mathematical operations the library GLM[2] was used.

Features

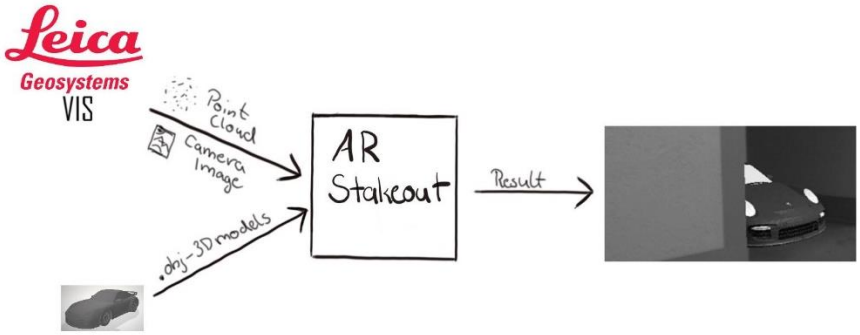
- Multi-pass rendering
  - each render-pass can be toggled at runtime
- Dynamic changes to the geometry
  - adding and removing meshes
  - movement
  - status (highlighting selected object)
- Blinn & Phong shading with directional light
- GUI using ImGui[3] with small adaptions to reuse Framebuffer
- Triple-Buffering
- Reloading of shaders during runtime to allow quick testing of shaders without restarting the application & reloading the geometry

Architecture

The main class is handling the setup of the Vulkan instance and device and holding the main loop. Render pass specific operations are done within the Shader pipeline module and an interop class handles conversion of data from VIS to GLM and application structures. This includes for example conversion of OpenCL matrices to GLM matrix types.

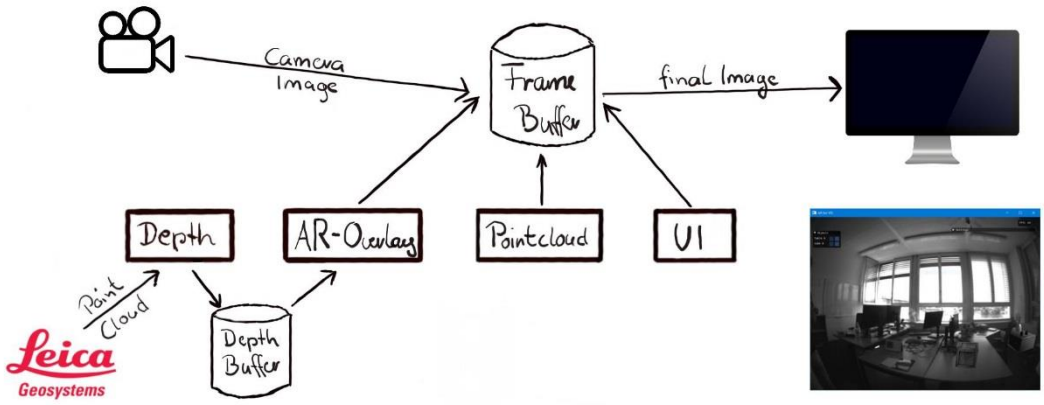
The main loop handles any changes done via the GUI and passes the required parameters along to the corresponding submodules, queues all enabled render passes with correct dependency settings in between them and finally presents the rendered image to the Window surface.

The input from Leica VIS, point cloud, camera images and position information for these, is combined with 3D objects placed by the user, ranging from simple primitives like cubes to complex models loaded from .obj files, to create a realistic AR image.



[1] Latest version at time of development is 1.1.77.0
[2] "OpenGL Mathematics."
[3] "Ocornut/Imgui: Dear ImGui: Bloat-Free Immediate Mode Graphical User Interface for C++ with Minimal Dependencies."

Rendering



The application uses multiple command-buffers and pipelines with shared frame-buffer images to render different parts of the scene.
The first operation is to load the camera image onto the framebuffer. This is done by bliting the source image without a mask. A blit operation is an image combination method using a boolean function for composition. In this case the OR function is used to overwrite the empty destination framebuffer image with the source data.
The reason to use a blit here instead of a simple copy, is that the Vulkan function '*vkCmdBlitImage(…)*' allows to specify the target region size, which will automatically stretch or downscale the image to fit the framebuffer extent.

As second stage, the point cloud, provided by the device and accessed using Leica VIS, is rendered into the depth buffer for correct occlusion. This stage does not write anything into the framebuffer.

The main render pass is the drawing of AR-overlays. This stage loads the frame- and depth-buffer with the values from the previous stages and draws its results on top of it. As the depth information is present from the point cloud, objects will be correctly culled in the depth & stencil stage of Vulkan's fixed function pipeline.

Color rendering of the point cloud is skipped by default. While the main intention of the point cloud is correct occlusion, this additional stage can be used to visually verify the position and density of the point cloud in the current scene. If enabled the point cloud is rendered with white color and an alpha value of 0.5 on top of the AR-Overlay output.

Finally, the UI elements created by ImGui[4] are drawn on top of everything and the framebuffer is presented on the application surface.

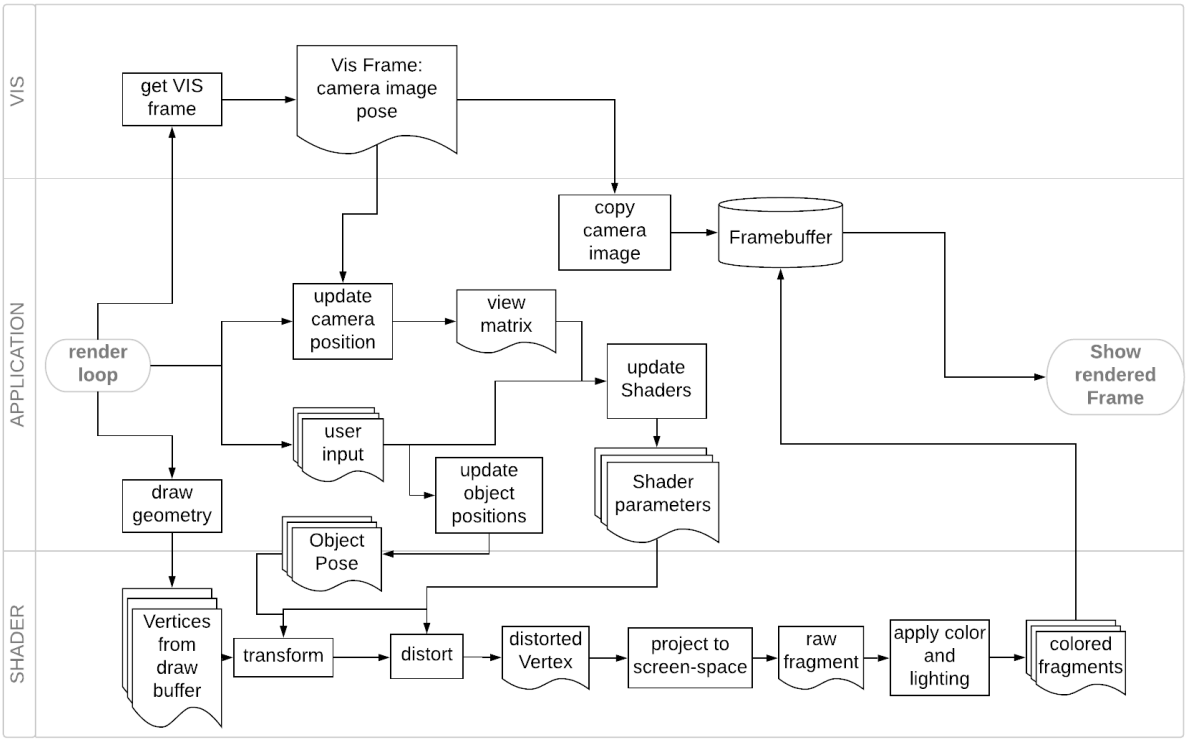
The dependencies are set-up in a way, that every one of the three main pipelines can be enabled or disabled during runtime, while only the initial clear of the frame- and depth-buffer images and rendering of the GUI overlay are required. The blit command is only executed if a valid input image is provided.

Sharing the frame-buffer images between all of these pipelines allows direct use of the previous stage's results without having to copy the data around. The only downside is that the

[4] "Ocornut/Imgui: Dear ImGui: Bloat-Free Immediate Mode Graphical User Interface for C++ with Minimal Dependencies."

stages that write to the color image cannot be executed in parallel. This is no issue for the depth- and AR pass, as those must run sequentially, because the AR pass depends on the depth information from the depth-pass.

AR-Overlay render pass



The diagram above shows the interaction between the application, Leica's VIS library and the AR-Pipeline. The application asks VIS to provide a Frame and use the Pose information to update the Camera's view matrix. The Frame image is sent to the first render-pass, which will blit[5] the content to the current frame-buffer image.
Changes the user made to the geometry via the UI are stored in the Dynamic Uniform Buffer, which contains per object model matrices and state flags. The other shader parameters like the view and projection matrices, light settings, etc. are stored in typical uniform buffers, that are reused on every object.
The AR shader pipeline transforms the vertices into view space, distorts the subdivided vertices in the tessellation evaluation stage and applies lighting and color in the fragment stage. By default, the fragments are converted to grayscale as the input image from the device is also grayscale. However, this can be changed in the UI.
The color output from the AR pipeline is stored in the framebuffer and passed to the next pipeline.

[5] "VkCmdBlitImage(3)."

Algorithmic

Virtual camera for input video

The virtual camera is a representation of the device camera providing the video stream with updated translation and rotation for every frame with valid position information. The frames and position information are provided by Leica VIS.
Having this representation of the moving camera allows objects placed relative to the application's world coordinate system to retain their position in regards to the camera image. The point cloud's coordinate system is used as the world system, which means that point

$$p^W = p^S = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} T$$

$W = world\ coordinate\ system$
$S = sensor\ coordinate\ system$

is at the origin of the point cloud.
The image cameras are not the same as the sensor from which the point cloud is created. Each has a positional and rotational offset relative to the device, which, together with the pose (position and rotation) for a frame, is used to calculate the view matrix of the virtual camera

Position

$$t_C^W = t_s^W + \left(R_s \ * \ t_c^D\right)$$

$D = device\ coordinate\ system$
$C = camera$
$S = sensor$
$t_C^W = position\ of\ camera\ in\ world\ coordinates$
$R_S = rotation\ matrix\ of\ sensor$

Rotation

The application is operating in the world coordinate system which equals the point cloud's coordinate system. However, camera poses are in the device coordinate system, which is rotated in -90 degrees around the x-Axis in regards to the world coordinate system.
Therefore multiplying the Rotation matrix R_{DtoW}, which describes a positive 90 degree rotation around the x-Axis, to the camera's local pose rotation matrix transforms the device local poses to world coordinates. The resulting rotation matrix for the camera is

$$R_{VC} = R_s \ * R_{DtoW} * R_c$$

$R_{VC} = rotation\ matrix\ of\ virtual\ camera$

View Matrix

Combining the translation and rotation from the previous calculations results in following view matrix:

$$View = \begin{pmatrix} R_{11} & R_{12} & R_{13} & t_1 \\ R_{21} & R_{22} & R_{23} & t_2 \\ R_{31} & R_{32} & R_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective Projection

For correct perspective and projection, the virtual camera's projection matrix has to be built according to the intrinsic parameters from the camera calibration. The calibration data is available from VIS and the projection matrix built by following formula, according to these sources[6][7][8]

$$Proj = \begin{pmatrix} \frac{f_x}{w} & 0 & \left(1 - \frac{p_x}{w}\right) & 0 \\ 0 & \frac{f_y}{h} & \left(\frac{p_y}{h} - 1\right) & 0 \\ 0 & 0 & -\frac{Z_f + Z_n}{Z_f - Z_n} & -2 * (\frac{Z_f * Z_n}{Z_f - Z_n}) \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$f = focal\ length$ $\quad p = principal\ point$
$Z_n = near\ plane$ $\quad Z_f = far\ plane$
$w = \frac{width^{Image}}{2}$ $\quad h = \frac{height^{Image}}{2}$

Note:
As the Vulkan coordinate system has the y-axis pointing downwards, $Proj_{11}$ is multiplied by -1. Alternatively, the position emitted from the vertex shader could also have the y value inverted with gl_Position.y *= -1.0;

[6] "Camera Calibration and 3D Reconstruction — OpenCV 2.4.13.7 Documentation."
[7] "Guillaume Chereau Blog - OpenCV Camera to OpenGL Projection."
[8] "Calculating OpenGL Perspective Matrix from OpenCV Intrinsic Matrix."

Distortion on the GPU

The camera used for this project has a heavy higher-order distortion. The distortion model is based on the OpenCV camera calibration[9] and implemented in the tessellation evaluation shader.
The implementation is an adapted version of the distortion formula from OpenCV, translated to GLSL code.

$$p = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$
$$x' = \frac{x}{z}$$
$$y' = \frac{y}{z}$$
$$r^2 = x'^2 + y'^2$$
$$d^{radial} = \frac{1 + \left(\left(\left((k_6 * r^2 + k_5) * r^2 + k_4\right) * r^2 + k_3\right) * r^2 + k_2\right) * r^2 + k_1\right) * r^2}{1 + \left((k_3 * r^2 + k_2) * r^2 + k_1\right) * r^2}$$
$$s^{tangential} = 1 + \left((t_6 * r^2 + t_5) * r^2 + t_4\right) * r^2 + t_3\right) * r^2$$
$$d_x^{tangential} = \left(2 * t_1 * x' * y' + t_2 * (r^2 + 2 * x'^2)\right) * s^{tangential}\right) + shear * x' + skew * y'$$
$$d_y^{tangential} = t_1 * (r^2 + 2 * y'^2) + 2 * t_2 * x' * y' * s^{tangential}$$
$$x'' = x' * d^{radial} + d_x^{tangential}$$
$$y'' = y' * d^{radial} + d_y^{tangential}$$

As the distortion is applied after z-division, but before viewport translation, the shader distorts vertices after model-view transformation and applies a temporary z-division. Therefore, in the shader x' and y' are multiplied with their original z-value after distortion and before multiplication with the projection matrix.
The advantages of doing distortion in the tessellation shader vs. doing it in the vertex stage is very well visible when objects with a low vertex count, for example a cube with only 8 vertices. The edges of the object will always be straight, as the distortion is working on the vertex and edges are generated afterwards. This problem is solved by using tessellation to subdivide each triangle into more triangles (number depends on the selected tessellation level), which means that in the case of an edge between v_1 and v_2 there now are $v_1, v_2, \ldots, v_n$ vertices. The edges between the individual vertices are still straight, but as the individual vertices from the subdivision are distorted, the result gets closer to a curved edge with higher tessellation levels.

3 no distortion

2 distortion in tessellation shader

[9] "Camera Calibration and 3D Reconstruction — OpenCV 2.4.13.7 Documentation."

Removing artifacts by using undistortion mask

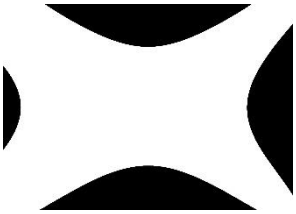
The higher order terms in the distortion model lead to a non-monotonic reprojection of 3D points. This causes points that are actually outside the field-of-view to become visible in the rendering. There are 2 checks implemented to prevent this from happening.

The first check is to make sure, that the point which should be distorted is a part of the visible image. However, the camera image is also distorted and therefore a typical frustum check does not provide good results but leads to objects that are close to the edges of the image being cut away.
Since the standard pinhole reprojection is monotonic with respect to the undistorted image plane, the visibility check must be done in undistorted coordinates. Therefore, we apply the undistortion mask, which marks areas in the undistorted image that are not valid.

Aquisition of undistortion mask

Distorted Input Image

Undistorted input image



resulting mask

To acquire an undistortion mask at first the input image from the sensor is undistorted according to the cameras' distortion model. The resulting undistortion mask is a variant of the undistorted image where all pixels that contain image information of the distorted image are mapped to 1 and the rest to 0.

By passing this as a sampler to the shader the algorithm for checking if vertices should be distorted is quite trivial:

$$vert^{clip} = (x, y, z, w) \qquad \text{[-w, w]}$$
$$vert^{NDC} = \frac{vert_{xyz}}{vert_{w}} \qquad \text{[-1, 1]}$$
$$vert^{screen} = \frac{vert_{xy}^{NDC}}{2} + \frac{1}{2} \qquad \text{[0, 1]}$$

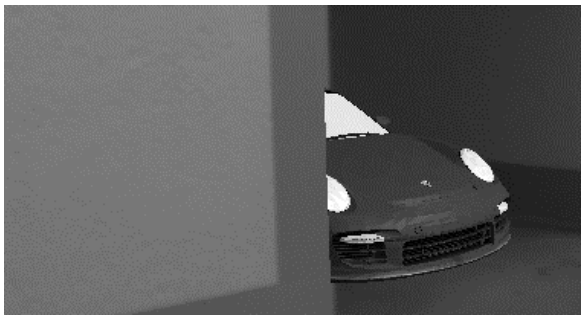
A simple texture lookup at $vert^{screen}$ in the red channel of the undistortion-mask sampler results in a value >0 if it is part of the undistorted image. Vertices that are outside of the undistortion-mask are set to $v = (2, 2, 2, 1)$ to cull them at the rasterization stage and save a bit of computing time. There is also an implementation using the fragment shader operation "discard" which tells the GPU to not draw the current fragment. However, this is more computationally intense and also the operation seems to not be supported on Intel GPUs and lead to a "Integer Division by Zero" exception in the Intel driver during pipeline initialization.

Occlusion

An important trait of realistic AR is correct occlusion with the environment. This leads to the perception of objects blending well into the scene.
The camera images are projections of the 3D world into the camera sensor plane, which leads to a loss of depth information. However, the Leica device used for this project provides a 3D point cloud along every axis. This point cloud can be used to reconstruct the 3D position of an arbitrary image point and, more importantly, with small tricks it can provide a dense depth buffer for occlusion culling using the GPU's depth testing.

The point cloud can be interpreted as a 3D object, where each point of the cloud is treated as a vertex and transformed with the current camera's view and projection matrixes. As the point cloud is centered around $p = (0,0,0)$, the model-matrix of this point-cloud object is always the identity matrix.
The result is the depth information for the visible area of the camera with regards to the current position and rotation.
The render-pass for AR overlays now has to load the depth-buffer and enable depth testing on render-pass creation to allow the GPU to cull the rendered objects with the fixed function depth-testing.

5 without depth information from pointcloud



4 pointcloud rendered to Z-Buffer

In the current version of the application, the point cloud provided by Leica VIS is not meshed. Therefore it does not provide a dense depth buffer by default. To work around this the point cloud shader is using an increased point size of 3.0 to approximate a dense mesh. This works reasonably well for the scope of this early pre prototype and environments from half a meter to around 10m.

However, for further development it is planned to generate a mesh from the point cloud to accurately calculate the depth buffer for a wider range of environments.

Conclusion

The main goal of the project was reached, by developing a rendering framework for AR applications using Vulkan with integration of Leica's VIS to provide a camera stream with pose information at camera rate (15 Hz).
It is possible to query the 3D position for arbitrary points in the image (with the help of the point cloud information).

One of the main features of the framework is the virtual camera model built from physical camera parameters (intrinsic and extrinsic), which will accurately follow the positional and rotational changes of the real device. Moreover, the camera model performs correct projection regarding the physical camera's traits and the application specific settings such as clip distance and resolution.

Distorting the objects with regards to the camera distortion parameters (intrinsic) at the tessellation stage and the usage of depth information from the point cloud for occlusion culling results in objects blending well into the camera image to a point where they are only barely distinguishable.
Doing the distortion calculation on GPU is also very efficient compared to running it on the CPU. (FPS drop on Intel iGPU < 1.5%).

While offering generally quite good results, the approach to use a point cloud for occlusion culling is not perfect. Changes in the environment between point cloud generation and usage in the application lead to occlusion artefacts, where objects can be falsely culled, because there was an obstacle there before but moved away after, or objects are not occluded by obstructions added to the scene later.

Rendering the full point cloud at every frame is also not very efficient, and for further development should be converted to a triangulated mesh, preferably with view frustum culling implemented.

“Calculating OpenGL Perspective Matrix from OpenCV Intrinsic Matrix.” Accessed August 30, 2018. http://kgeorge.github.io/2014/03/08/calculating-opengl-perspective-matrix-from-opencv-intrinsic-matrix.
“Camera Calibration and 3D Reconstruction — OpenCV 2.4.13.7 Documentation.” Accessed August 30, 2018. https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.
“Guillaume Chereau Blog - OpenCV Camera to OpenGL Projection.” Accessed August 30, 2018. https://blog.noctua-software.com/opencv-opengl-projection-matrix.html.
“Ocornut/Imgui: Dear ImGui: Bloat-Free Immediate Mode Graphical User Interface for C++ with Minimal Dependencies.” Accessed August 30, 2018. https://github.com/ocornut/imgui.
“OpenGL Mathematics.” Accessed September 27, 2018. https://glm.g-truc.net/0.9.9/index.html.
“VkCmdBlitImage(3).” Accessed September 27, 2018. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/man/html/vkCmdBlitImage.html.