Integration of Image Processing Techniques into the Unity Game Engine

Leon Masopust[1], Sebastian Pasewaldt[2], Jürgen Döllner[1] and Matthias Trapp[1][a]
[1] *Hasso Plattner Institute, Faculty of Digital Engineering, University of Potsdam, Germany*
[2] *Digital Masterpieces GmbH, Potsdam, Germany*
{*leon.masopust, juergen.doellner, matthias.trapp*}@*hpi.de, sebastian.pasewaldt@digitalmasterpieces.com*

Keywords: Image Processing, Game Engines, Unity, Integration Approaches

Abstract: This paper describes an approach for using the Unity game engine for image processing by integrating a custom GPU-based image processor. For it, it describes different application levels and integration approaches for extending the Unity game engine. It further documents the respective software components and implementation details required, and demonstrates use cases such as scene post-processing and material-map processing.

1 INTRODUCTION

Game Engines (GEs) such as the Unity or the Unreal Engine, gain increasing popularity in Computer Science with respect to teaching, training, or for prototyping interactive visualization techniques. Besides the potential support for multiple target platforms, GEs offer basically a technology platform used for performing feasibility studies, performance tests, rapid application development, and to generate test data for the development of new techniques (Lewis and Jacobson, 2002).

Using such functionality enable faster iterations on prototypes and support simplification of the overall implementation process. Thus, one major advantage of using GEs as a integration and implementation platform is rapid prototyping, i.e., to effectively perform feasibility studies or for teaching purposes. Further, GEs can be considered a base system for integrating 3D rendering to multiple back-ends and Virtual Reality (VR) applications. Generally, it lowers the technical barrier as well minimize project setup and deployment costs and thus enable practitioners and students to focus on algorithm technique development instead of engineering tasks.

Although previous work exist, that uses Unity as a platform for image processing (de Goussencourt and Bertolino, 2015; Anraku. et al., 2018), possible integration approaches and its implementation and implications are not covered so far. For a feasibility study, the paper uses Unity as an example for a 3D GE, however, the described concepts are not limited to. We choose Unity because, it represents a popular and a common-used real-time GE that is freely available and well documented. It is easily extendable at different levels (Section 3.2), e.g., using native rendering plugins. Further, it potentially supports multiple platforms and provides sophisticated Augmented Reality (AR) and VR components and interfaces.

This paper presents the methodology and application examples for performing image and video processing techniques within real-time GEs using dedicated hardware support, such as Graphics Processing Units (GPUs). Therefore, it presents a concept for integrating 3rd-party image processors into Unity and discusses the results by means of different application examples. To summarize, this paper makes the following contributions:

1. It describes and discusses different application levels as well as suitable integration strategies for 3D GEs.
2. It reports on a feasibility study regarding the Unity GE and different Visual Computing Assets (VCAs) (Dürschmid et al., 2017).
3. It demonstrates our approach using different application examples.

The remainder of this paper is structured as follows. Section 2 reviews related and previous work with respect to using GEs in scientific contexts. Section 3 describes the concept and implementation of integrating a 3rd-party image and video processing component into the Unity GE. Based on that, Section 4 presents and discusses different case studies and present ideas for future research directions. Finally, Section 5 concludes this work.

[a] https://orcid.org/0000-0000-0000-0000

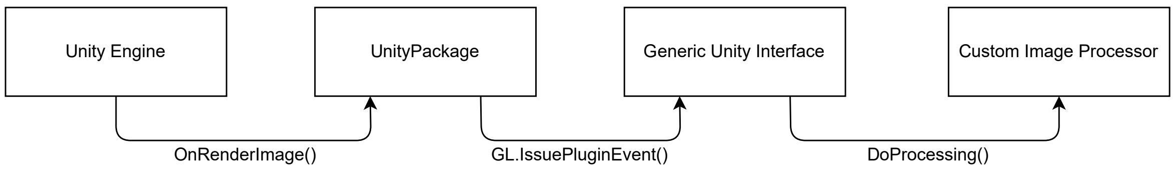

Figure 1: Overview of a basic integration approach of a 3rd-party image processor into the Unity GE.

2 RELATED WORK

In recent work, game engines are often used for prototyping interactive visualization techniques and for educational purposes in 3D Computer Graphics (CG). This section focuses on research results that use GEs in scientific work, tools, and integration approaches.

Game Engines in Visualization. In the past, GE were often used for innovations and prototyping interactive visualization (Rhyne, 2002; Wagner et al., 2016) and pre-visualization (Nitsche, 2008). With respect to geovisualization, Herwig and Paar discuss options and limitations of GEs as low-priced tools for landscape visualization and planning (Herwig and Paar, 2002). They argue and demonstrate that particular software components are useful alternatives to landscape architects and planners, e.g., to support collaborative landscape planning, although a number of professional features are not supported. Further, Fritsch and Kada emphasis the usefulness real-time visualization using GEs with respect to presentation purposes in the domain of Geographic Information Systems (GIS) and Computer Aided Facility Management-Systems (CAFM) (Fritsch and Kada, 2004), especially considering the possibilities for mobile devices such as notebooks and cell phones. They show that the integration of additional data and functionality into such systems can be achieved by extending the internal data structures and by modifying the accompanying dynamic link libraries.

Andreoli *et al.* provide a categorization of 3D GEs regarding their usage in creating interactive 3D worlds and a comparison of the most important characteristics (Andreoli et al., 2005) regarding interactive archaeological site visualization. In addition thereto, Wünsche *et al.* analyze the suitability of GEs for visualization research in general and present a software architecture and framework for a data transformation and mapping process to facilitates their extension as well as further evaluate the suitability of popular engines respectively (Wünsche et al., 2005). A similar approach and comparison was conducted by Kot *et al.* targeting the domain of information visualization (Kot et al., 2005). In the domain of software visualization, Würfel *et al.* extend the Unreal GE to prototypical implement metaphors for visualization of trend data in interactive software maps (Würfel et al., 2015).

More recently, Bille *et al.* present an approach to visualize Building Information Model (BIM) data using a GE (Bille et al., 2014). Their case study demonstrates the conversion from BIM to GEs from the BIM tool Revit to the Unity game engine. Similar thereto, Ratcliffe uses GEs to create photo-realistic interactive architectural visualizations (Ratcliffe and Simons, 2017). Specific to terrain visualization, Mat *et al.* present a review of 3D terrain visualization techniques using game engines (Mat et al., 2014).

In addition to real-time rendering capabilities, Jacobson and Lewis emphasis the importance of GEs for VR applications (Jacobson and Lewis, 2005). They use the Unreal GE as a low cost alternative to construct a CAVE installation. Similar thereto, the approach of Lugrin *et al.* relies on a distributed architecture to synchronize the user's point-of-view and interactions within a multi-screen installation in real-time (Lugrin et al., 2012). An accompanying user study also demonstrates the capacity of GE's VR middleware toelicit high spatial presence while maintaining low cybersickness effects. There is also a number of recent work that focus on other GE aspects besides rendering. For example, Juang *et al.* use GEs for physics-based simulations (Juang et al., 2011) while Leahy and Dulay present cyber-physical platform for crowd simulation (Leahy and Dulay, 2017).

Education using Game Engines. Another application field of GEs is education. For example, Marks *et al.* evaluated GEs for simulated surgical training, concluding that these represent a good foundation for low cost virtual surgery applications and identified limitations of physical simulation capabilities (Marks et al., 2007). Targeting high-resolution displays, Sigitov *et al.* present an extension of Unity that allows for the implementation of applications that are suitable to run on both single-display and multi-display systems (Sigitov et al., 2015). This approach simplifies software development, especially in educational context where the time that students have for their projects is limited.

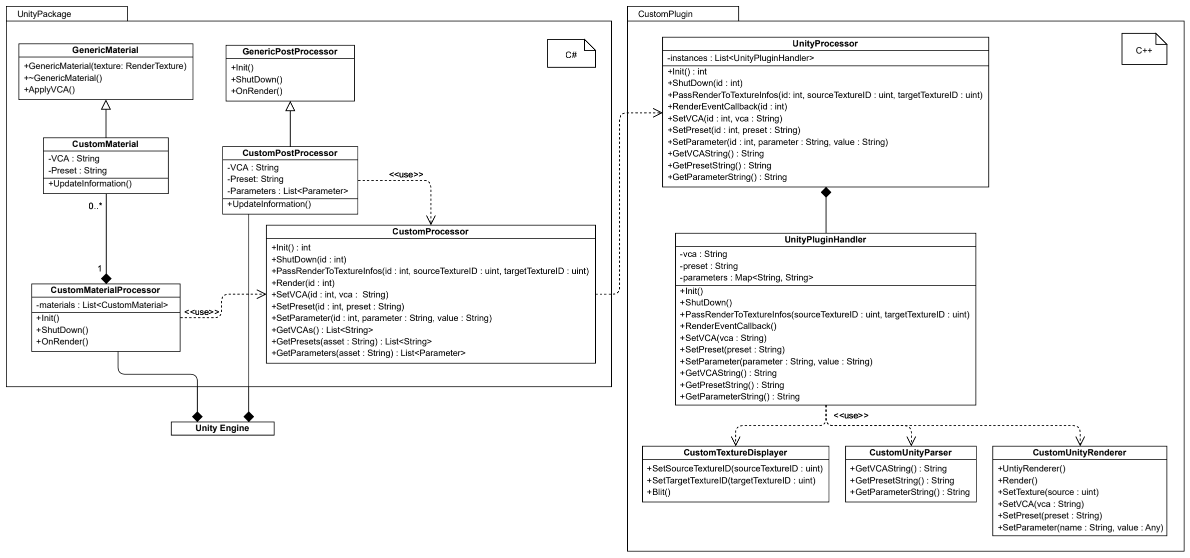
UnityPackage
C#
GenericMaterial
+GenericMaterial(texture: RenderTexture)
+~GenericMaterial()
+ApplyVCA()
GenericPostProcessor
+Init()
+ShutDown()
+OnRender()
CustomMaterial
-VCA : String
-Preset : String
+UpdateInformation()
CustomPostProcessor
-VCA : String
-Preset: String
-Parameters : List<Parameter>
+UpdateInformation()
<<use>>
0..*
1
CustomProcessor
+Init() : int
+ShutDown(id : int)
+PassRenderToTextureInfos(id : int, sourceTextureID : uint, targetTextureID : uint)
+Render(id : int)
+SetVCA(id : int, vca : String)
+SetPreset(id : int, preset : String)
+SetParameter(id : int, parameter : String, value : String)
+GetVCAs() : List<String>
+GetPresets(asset : String) : List<String>
+GetParameters(asset : String) : List<Parameter>
CustomMaterialProcessor
-materials : List<CustomMaterial>
+Init()
+ShutDown()
+OnRender()
<<use>>
Unity Engine
CustomPlugin
C++
UnityProcessor
-instances : List<UnityPluginHandler>
+Init() : int
+ShutDown(id : int)
+PassRenderToTextureInfos(id: int, sourceTextureID : uint, targetTextureID : uint)
+RenderEventCallback(id : int)
+SetVCA(id : int, vca : String)
+SetPreset(id : int, preset : String)
+SetParameter(id : int, parameter : String, value : String)
+GetVCAString() : String
+GetPresetString() : String
+GetParameterString() : String
UnityPluginHandler
-vca : String
-preset : String
-parameters : Map<String, String>
+Init()
+ShutDown()
+PassRenderToTextureInfos(sourceTextureID : uint, targetTextureID : uint)
+RenderEventCallback()
+SetVCA(vca : String)
+SetPreset(preset : String)
+SetParameter(parameter : String, value : String)
+GetVCAString() : String
+GetPresetString() : String
+GetParameterString() : String
<<use>>
CustomTextureDisplayer
+SetSourceTextureID(sourceTextureID : uint)
+SetTargetTextureID(targetTextureID : uint)
+Blit()
CustomUnityParser
+GetVCAString() : String
+GetPresetString() : String
+GetParameterString() : String
CustomUnityRenderer
+UntiyRenderer()
+Render()
+SetTexture(source : uint)
+SetVCA(vca : String)
+SetPreset(preset : String)
+SetParameter(name : String, value : Any)

Figure 2: Class diagram of the static system architecture. Inside the `UnityPackage` resides classes that are available and configurable inside the Unity GE and used together with engine hooks, e.g., `OnRenderImage()`. The classes `CustomPlugin` are compiled to a native library. Only the C interface of `UnityProcessor` is visible to the engine. The other classes implement typical used functions for the `UnityPackage` using an existing image processor framework.

3 GAME ENGINE INTEGRATION

This section describes different Application Levels (AL) for GEs (Section 3.1), discussing possible Integration Approachs (IAs) (Section 3.2), and presents the conceptual and technical aspects of an Unity integration approach (Section 3.3).

3.1 Application Levels

The integration of a real-time image processor into a GE can be performed at the following AL:

Static Material-Processing (AL-1): This refers to the preprocessing of material texture map (e.g., representing albedo, height, or normal maps) in the design phase prior to entering the game loop. Using this, game designer or technical artists can test different parameters settings for filter operations prior to final processing or asset baking.

Dynamic Material-Processing (AL-2): To support dynamic parameter changes for material texture maps (e.g., masks controlled by animations or via scripting) at runtime during game loop execution, dynamic material-processing can be used. Using this, the material appearance can be adapted interactively and multiple objects with different materials can be displayed simultaneously, e.g., to compare variations of an VCA.

Scene Post-Processing (AL-3): With respect to integration, this can be considered as a special case of dynamic material processing. It applies image-processing operations as a post-processing operations on a per-frame basis for virtual cameras.

Supporting these ALs facilitates interactive parameters settings with respect to the virtual camera and the texture material used (input image and videos).

3.2 Integration Approaches

Performing image and video processing in game engines can be achieved by the following three different Integration Approachs (IAs). For summarization, Table 1 shows a comparison between these.

Native GE Tooling (IA-1): This integration approach uses native GE functionality and tools for enabling image or video processing requires the analysis, design, and implementation of the frameworks core components. With respect to Unity, this can be achieved using scripting and the Unity Shader Graph component. However, this would require to re-implement existing

Table 1: Overview of Integration Approach (IA) and covered Application Levels (AL).

Approach	**AL-1**	**AL-2**	**AL-3**
IA-1		○	○
IA-2			○
IA-3	○	○	○
IA-4	○	○	○

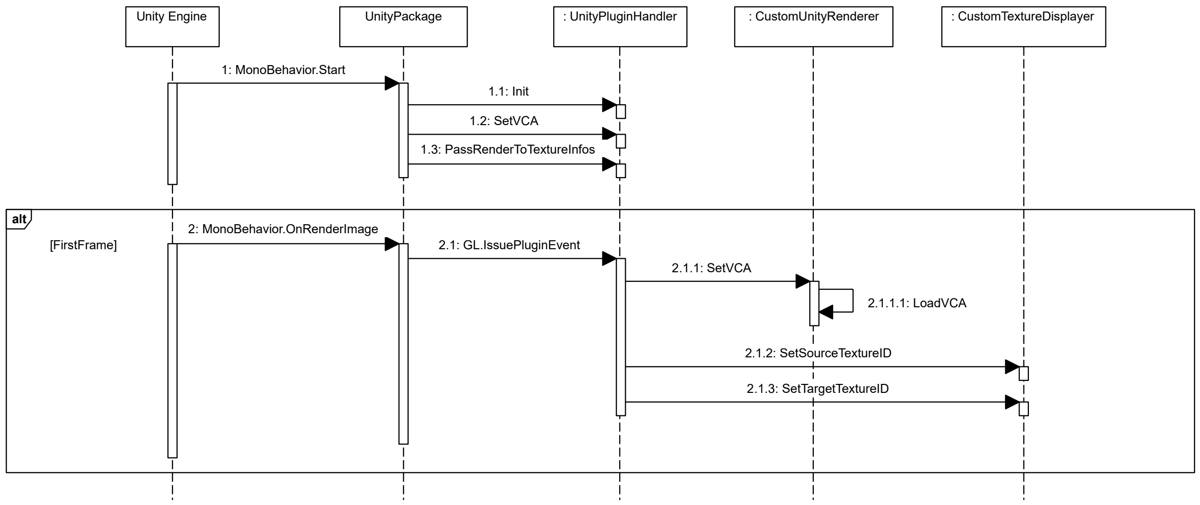

Figure 3: Initialization phase of the plug-in. Since the usage of graphic device functions are limited to special render contexts inside the engine (`GL.IssuePluginEvent`), the initialization phase is divided into a data transfer phase and the real initialization phase where the saved data is applied with the use of graphic device specific code.

functionality (e.g., , shader programs and control flow logic), which impact reusablity as well as increases development efforts.

GE Framebuffer Output (IA-2): This approach uses the GE rendering raster output as input for the 3rd-party library. This can cause heavy drawbacks in terms of performance, due to the copy of framebuffer data involved. However, often only Red-Green-Blue (RGB) data is received, thus no further Unity data can be used (e.g., normal or material maps) without additional G-Buffer (Saito and Takahashi, 1990) implementation support.

GE Patching (IA-3): If the source code of a GE is available and licenses allow it, this integration approach extend the GE functionality by patching. This enables the development of missing features directly into the GE. However, this approach impacts often software maintenance aspect.

GE Plug-In Development (IA-4): For GE that sufficiently supports plug-in interfaces or Software Development Kits (SDKs), integration can be conducted by using these to interface 3rd-party software components – given a respective Application Binary Interface (ABI) compatibility. This integration approach combines flexibility and code re-usability, as well as enables deployment to possibly different platform. This way, one can benefit from multiple GE feature and components, such as User Interfaces (UIs), prototype modes, tools for performance measurements and optimizations. If thoughtfully conducted, an additional performance impact can be neglected. However, to implement such an approach, an understanding of the GE internals is required and thus initial steps can be cumbersome and error-prone.

In order to implement IA-4, this paper presents the necessities and details in the following section. This can serve as template for the integration of similar software components into Unity.

3.3 Implementation Details

Native rendering plug-ins in Unity are written and compiled platform-specific and need to provide a C interface to communicate with components inside the GE. To support multiple platforms and multiple graphic device interfaces, distinct implementations are required To reduce the implementation workload, a generic interface was created. The classes `CustomTextureDisplayer`, as well as `CustomUnityParser` and `CustomUnityRenderer` (Figure 2) encapsulate functions, which are used inside the engine. Every new image processing framework or platform specific code can implement the interface functions to achieve an integration into the plug-in.

As a proof-of-concept instantiation, we use the Windows platform and OpenGL for evaluation, since the existing image processor framework already had support for this combination. Plug-in function are mainly called during initialization phase (Section 3.3.1) and the actualy rendering loop (Section 3.3.2).

3.3.1 Initialization Phase

During the initialization phase, all information required by the image and video processor are trans-

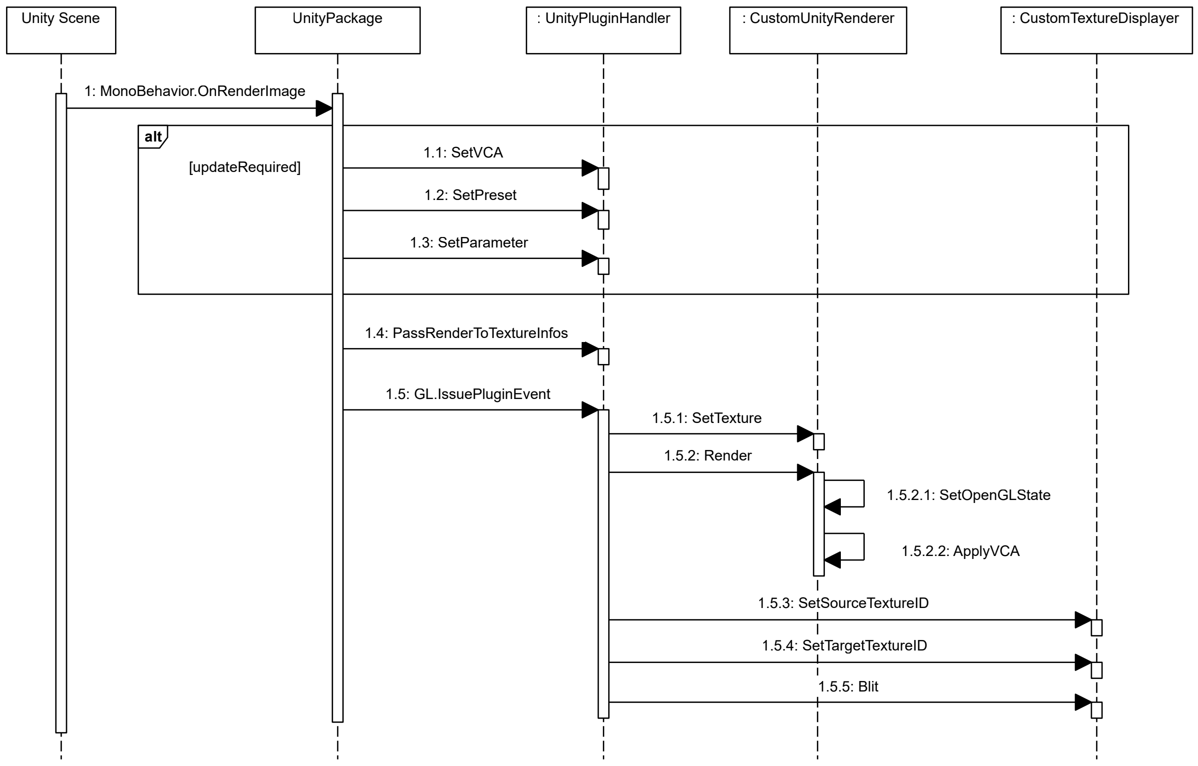

Figure 4: Handling per-frame of the plug-in. If required data such as texture Identifiers (IDs) are updated and saved inside the plug-in. Together with the render hook, the image processor applies the VCA directly onto the texture and blits it to the output texture, which is determined inside the engine.

ferred to the native rendering plug-in. While this is easily possible during the `OnEnable()` or `Start()` hook in Unity, the graphic device specific initialization steps of the image processor (texture, vertex buffer setup, etc.) require the same active context that is used during the rendering. Due to this, all transferred information are only stored inside the plug-in, e.g., with `SetVCA()` or `PassRenderToTextureInfos()` (Figure 3), and then used during the first frame of the rendering loop to load the VCA or to encapsulate the native texture IDs.

3.3.2 Rendering Loop

To integrate into Unity's rendering pipeline, multiple approaches (hooks) are possible. In case of post-processing (AL-3), the image processor needs to work with the final framebuffer content and thus is called inside an `OnRenderImage` hook of a Unity post-processing effect. For the material processing it is important to have the image and video processor finished before any rendering starts. Hence it is called inside the `OnPreRender` hook of the first used virtual camera. Inside these hooks, Unity provides the possibility to add a rendering callback (`GL.IssuePluginEvent()`) which is always executed asynchronously, but always prior to the next rendering step. This can lead to major problems, when a synchronous behavior is required. For example, when changing the VCA, the new configuration should be directly queried to be visualized inside the Unity UI, therefore an active render context is required. Thus a combination of using rendering callbacks and busy waiting was implemented to ensure the rendering specific functions were finished.

During rendering, it is important to always handle changes accordingly. Besides domain specific parameters, such as VCAs or presets, rendering-specific parameters might change too, which can result in graphic device errors or even crashes if not handled properly. Those rendering-specific parameters can be texture sizes or even native ID switches, if the game engine decides to cycle between framebuffers for optimizing multiple post-processing effects. On the plug-in side, these changes are effectively managed by using cache-based structures to store the native ID encapsulations and VCAs. Afterwards, the desired OpenGL State needs to be established (`SetOpenGLState()`) to get the image processor to work and the selected VCA can be applied onto the texture with the rendered Unity content, which is directly done on GPU without trans-

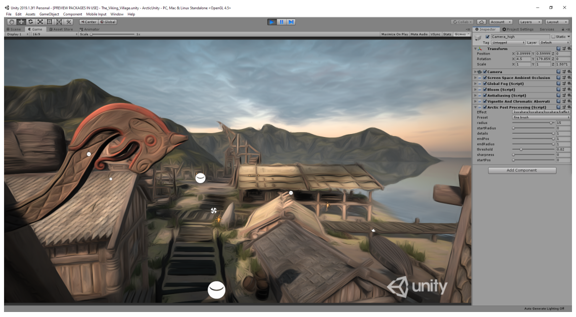
Figure 5: Screenshot of the scene post-processing case study implementation as image effect in the Unity. A Kuwahara filter is applied as last filter to the scene similar to other build-in effects such as Bloom or Antialiasing. The parameters are customized according to the lighting and atmosphere of the scene.

ferring any framebuffer data (`ApplyVCA()`, Figure 4). For the last step, the output texture of the image processor is blit to the target texture determined inside of Unity (next framebuffer or even the same texture in case of material processing).

4 RESULTS & DISCUSSION

This section describes and discusses two exemplary uses case that can be put into practice using the presented approach (Section 4.1 and Section 4.2). Based on these case-studies, our approach is discussed with respect to its usability (Section 4.4) and it run-time performance overhead (Section 4.3).

4.1 Scene Post-processing

Post-processing VCAs for real-time rendering are fully integrated into Unity and can be controlled from script through the `OnRenderImage` rendering hook (AL-3). Furthermore, already built-in effects such as Antialiasing or Bloom are exposing parameters to making VCA customization easier.

The custom image processor for post-processing is also implemented as camera script and gives users the possibility to change their VCA, preset, or parameters (Figure 5). The image processor receives the framebuffer content after the scene is completely rendered, then manipulates it, and hands it over to the next post-processing stage or to the output. Thus, it can be freely combined with other custom image processors or build-in post-processing. Applications can be manifold, such as (1) compensating color blindness, (2) highlighting interesting parts of the scene, or (3) giving the application an unique artisitic style.

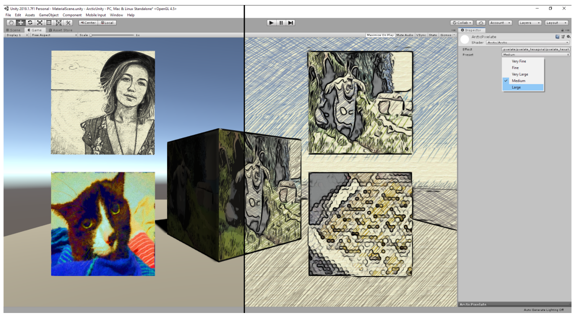
Figure 6: Screenshot of the material-map processing in the Unity Editor. Effect and preset are selected for the render texture in the bottom right. The material processing can be combined with custom post-processing effects (as shown in the right part of the image).

4.2 Material-Map Processing

In contrast to post-processing VCAs, material VCAs are not an already integrated in Unity 3D ((AL-1 and (AL-2)). Nevertheless they are a suitable to customize scene objects in a more dynamic and advanced way.

The material processing is devided into two parts (Figure 2): the `CustomMaterial` can be used on materials, which then are applied on scene objects and allow users to select a effect and preset (Figure 6). The `CustomMaterialProcessor` contains all `CustomMaterials`, which are applied to objects in the scene. Depending on the render mode of the `CustomMaterial`, the processor either applies the effect once on start or every frame inside the `OnPreRender` hook. Since the material processing is implemented for render textures, it can be used in various ways, such as static material enhancement or dynamic material manipulation.

The presented approach also enables the combined usage of both, material-map and scene post-processing. This allows for further customization and is especially applicable for applying post-processing effects on top of stylized materials (Figure 6).

4.3 Performance Evaluation

We evaluated the run-time performance of the integration approach using different algorithms. Different image resolutions were tested to estimate the run-time performance regarding the spatial resolution of an input image. The following resolutions were chosen: 1280×720 (HD), 1920×1080 (FHD), and 2560×1440 (QHD) pixels. We tested the rendering performance of our preliminary implementation (custom processor) using a NVIDIA GeForce GTX 970 GPU with 4096 MB VRAM on a Intel Xeon CPU with 2.8 GHz and 12 GB RAM. Rendering was performed in windowed mode with vertical synchroniza-

Table 2: Results of run-time performance measurements for VCA of different implementation complexity in Frames-per-Second (FPS) and milliseconds. It shows the timings obtained by rendering the 3D scene without any VCA (column 2) in comparison with VCAs applied (column 3). Further, it shows the runtime of processing the VCAs only (column 4) with the respective overhead resulting from the integration of our custom processor (column 5).

VCA	**Scene w/o VCA**	**Scene with VCA**	**VCA Only**	**Custom Processor**
Water Color (Bousseau et al., 2007)	99.38 (10.06 ms)	65.47 (15.27 ms)	191.85 (5.21 ms)	595.20 (1.69 ms)
Toon (Winnemöller et al., 2006)	99.38 (10.06 ms)	69.35 (14.42 ms)	229.50 (4.36 ms)	714.30 (1.42 ms)
Color LUT (Selan, 2005)	99.38 (10.06 ms)	75.65 (13.22 ms)	316.82 (3.16 ms)	1408.00 (0.71 ms)

tion turned off.

The measurements in FPS are obtained by averaging 500 consecutive frames. For all off-screen rendering passes, we use a fragment shader while rendering a textured Screen-aligned Quad (SAQ) with a geometric complexity of four vertices and two triangle primitives. For rasterization, back-face culling (Akenine-Möller et al., 2018) is enabled and depth test and writing to depth buffer are disabled. Table 2 shows the obtained run-time performance results. It shows, that the performance is mostly fill-limited. However, the integration overhead introduced by our custom processor is relatively small. Thus, complex stylization effects are not suitable for performance critical applications on modest graphics hardware.

4.4 Usability Aspects

During the case studies, we perform informal experts reviews, in order to evaluate the usability of our approach with respect to the stated goals. Participants were (1) software developers and (2) undergraduate/-graduate students in Computer Science with a focus on computer graphics systems.

In general, users liked the ease-of-use of the post-processing integration similar to other build-in effects in Unity. This is mostly due to reusing the build-in UI components of Unity. They further confirm that the post-processing integration is applicable for Rapid Prototyping (RP), and favor that material effects can be simply applied similar to the standard Unity approach. Especially, the possibility of using additional information such as scene depth, normal vectors, or texture masks as input in the custom processor were considered important.

However, some reviews also pointed out limitations and drawbacks of the integration, which can be approached in future work. Foremost, post-processing in general does not allow for masked areas for UI or text rendering, which makes compositing cumbersome. Further, the current integration approach does not allow to animate VCA parameters using build-in Unity animation functionality.

4.5 Future Research Directions

Based on the demonstrated results, there is are numerous potential research directions to approach. To extend material processing with parameter settings, the Unity core can be adapted to extend the material definition interfaces. Further, support for parameter animations at runtime can be enabled by making use of Unity's animation functionality. Furthermore, the described approach can be extended to adding Unity plug-in support for other image processing frameworks on Android or iOS devices.

5 CONCLUSIONS

This paper documents the software components and implementation details required for integrating a GPU-based image processing framework into the Unity game engine. We demonstrate the feasibility of the presented approach using two use cases that can be applied in combination: scene-based post-processing and dynamic material-processing. For it, we describe possible application levels and integration approaches and a performance evaluation, respectively. The described components and software architecture can serve as model or blueprint to integrate similar 3$^{\text{rd}}$-party custom image or video processing components into the Unity game engine.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work has been supported by the German Federal Ministry of Education and Research (BMBF) through grants 01IS15041 ("md-ViPro") and 01IS19006 ("KI-Labor ITSE").

REFERENCES

Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M., and Hillaire, S. (2018). *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA.

Andreoli, R., Chiara, R. D., Erra, U., and Scarano, V. (2005). Interactive 3d environments by using videogame engines. In *9th International Conference on Information Visualisation, IV 2005, 6-8 July 2005, London, UK*, pages 515–520.

Anraku., S., Yamanouchi., T., and Yanaka., K. (2018). Real-time integral photography holographic pyramid using a game engine. In *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 4: VISAPP,*, pages 603–607. INSTICC, SciTePress.

Bille, R., Smith, S. P., Maund, K., and Brewer, G. (2014). Extending building information models into game engines. In *Proceedings of the 2014 Conference on Interactive Entertainment*, IE2014, pages 22:1–22:8, New York, NY, USA. ACM.

Bousseau, A., Neyret, F., Thollot, J., and Salesin, D. (2007). Video watercolorization using bidirectional texture advection. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, pages 104–112, New York, NY, USA. Association for Computing Machinery.

de Goussencourt, T. and Bertolino, P. (2015). Using the Unity game engine as a platform for advanced real time cinema image processing. In *22nd IEEE International Conference on Image Processing (ICIP 2015)*, Québec, Canada.

Dürschmid, T., Söchting, M., Semmo, A., Trapp, M., and Döllner, J. (2017). Prosumerfx: Mobile design of image stylization components. In *Proceedings SIGGRAPH ASIA 2017 Mobile Graphics and Interactive Applications*, SA '17, pages 1:1–1:8, New York. ACM.

Fritsch, D. and Kada, M. (2004). Visualisation using game engines. *Archiwum ISPRS*, 35:B5.

Herwig, A. and Paar, P. (2002). Game engines: Tools for landscape visualization and planning. In *Trends in GIS and Virtualization in Environmental Planning and Design, Wichmann Verlag*, pages 161–172.

Jacobson, J. and Lewis, M. (2005). Game engine virtual reality with caveut. *Computer*, 38(4):79–82.

Juang, J. R., Hung, W. H., and Kang, S. C. (2011). Using game engines for physics-based simulations - a forklift. *Journal of Information Technology in Construction (ITcon)*, 16:3–22.

Kot, B., Wuensche, B., Grundy, J., and Hosking, J. (2005). Information visualisation utilising 3d computer game engines case study: A source code comprehension tool. In *Proceedings of the 6th ACM SIGCHI New Zealand Chapter's International Conference on Computer-human Interaction: Making CHI Natural*, CHINZ '05, pages 53–60, New York, NY, USA. ACM.

Leahy, F. and Dulay, N. (2017). Ardán: Using 3d game engines in cyber-physical simulations (tool paper). volume 10107, pages 61–70.

Lewis, M. and Jacobson, J. (2002). Game engines in scientific research. *Commun. ACM*, 45(1):27–31.

Lugrin, J.-L., Charles, F., Cavazza, M., Le Renard, M., Freeman, J., and Lessiter, J. (2012). Caveudk: a vr game engine middleware. In *Proceedings of the 18th ACM symposium on Virtual reality software and technology*, VRST '12, pages 137–144, New York, NY, USA. ACM.

Marks, S., Windsor, J., and Wünsche, B. (2007). Evaluation of game engines for simulated surgical training. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, GRAPHITE '07, pages 273–280, New York, NY, USA. ACM.

Mat, R. C., Shariff, A. R. M., Zulkifli, A. N., Rahim, M. S. M., and Mahayudin, M. H. (2014). Using game engine for 3d terrain visualisation of GIS data: A review. *IOP Conference Series: Earth and Environmental Science*, 20:012037.

Nitsche, M. (2008). Experiments in the use of game technology for pre-visualization. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, pages 160–165, New York, NY, USA. ACM.

Ratcliffe, J. and Simons, A. (2017). How can 3d game engines create photo-realistic interactive architectural visualizations? In *E-Learning and Games - 11th International Conference, Edutainment 2017, Bournemouth, UK, June 26-28, 2017, Revised Selected Papers*, pages 164–172.

Rhyne, T.-M. (2002). Computer games and scientific visualization. *Commun. ACM*, 45(7):40–44.

Saito, T. and Takahashi, T. (1990). Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206.

Selan, J. (2005). Using lookup tables to accelerate color transformations. In Pharr, M., editor, *GPU Gems 2*, pages 381–392. Addison-Wesley.

Sigitov, A., Scherfgen, D., Hinkenjann, A., and Staadt, O. (2015). Adopting a game engine for large, high-resolution displays. In *International Conference Virtual and Augmented Reality in Education (VARE)*.

Wagner, M., Blumenstein, K., Rind, A., Seidl, M., Schmiedl, G., Lammarsch, T., and Aigner, W. (2016). Native cross-platform visualization: A proof of concept based on the unity3d game engine. In *IV*, pages 39–44. IEEE Computer Society.

Winnemöller, H., Olsen, S. C., and Gooch, B. (2006). Real-time video abstraction. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 1221–1226, New York, NY, USA. Association for Computing Machinery.

Wünsche, B. C., Kot, B., Gits, A., Amor, R., and Hosking, J. (2005). A framework for game engine based visualisations. In *Image and Vision Computing*, New Zealand.

Würfel, H., Trapp, M., Limberger, D., and Döllner, J. (2015). Natural Phenomena as Metaphors for Visualization of Trend Data in Interactive Software Maps. In Borgo, R. and Turkay, C., editors, *Computer Graphics and Visual Computing (CGVC)*. The Eurographics Association.