UNREAL ENGINE

Image courtesy of Lune Rouge Entertainment

nDisplay Technology

Limitless scaling of real-time content

Contents

Introduction

All industries using real-time graphics on large displays have a common challenge of scaling and synchronizing real-time content on a wide variety of display media. Achieving success in this endeavor has been, to date, very challenging. The problems lie in a lack of sufficient processing power, communication difficulties between proprietary systems, and the need to make content that will play back in real time at sufficient speed.

To address these industry-wide problems, Epic Games researched and considered possible solutions for scaling real-time content. These efforts led to the development of the nDisplay system, which works with Unreal Engine to render 3D content simultaneously to multiple displays in real time.

In this document, we will look at the research behind the design and development of nDisplay technology, and how this research led to features that address these issues. We will provide an overview of currently available technology, current limitations, and plans for future development.

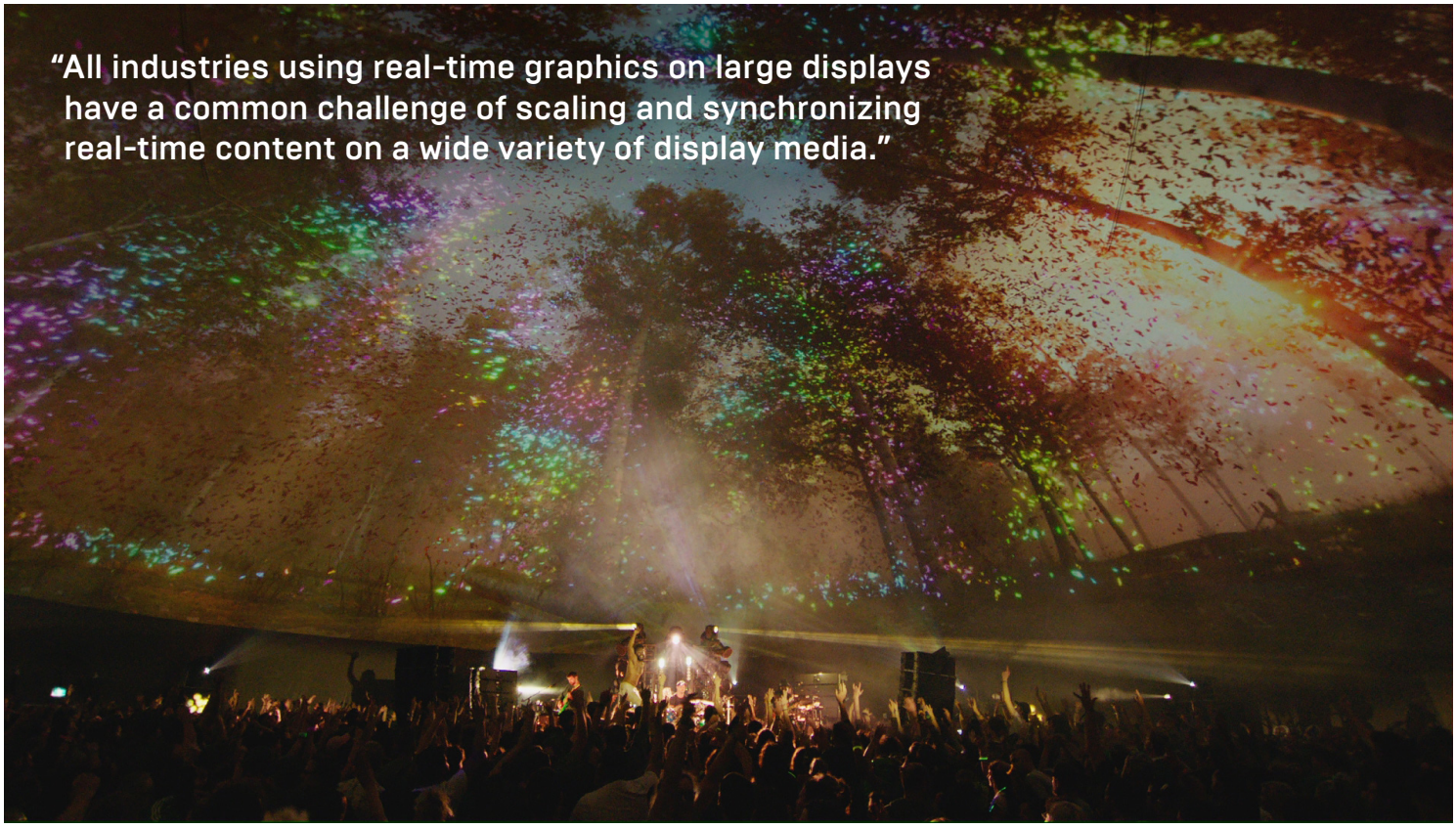

Image courtesy of Wolf + Rothstein

Background

Ever since computer graphics were introduced into the media world, there has been a desire and requirement to scale rendering capabilities in terms of increased resolution and reduced processing time. One approach has been to distribute rendering tasks over network clusters of multiple PCs ("render farms"), or multiple processor cores and software threads.

Then came the era of games and real-time graphics. Such graphics must be rendered at a rate of about 16 ms/frame to achieve a playback speed of at least 60 fps, the accepted rate for perceived realism. The development of dedicated graphics processing units (GPUs) has gone a long way toward achieving this.

However, much of this development has focused on increasing overall power through the addition of processing cores and available onboard memory, while increasing the speed at which they run. In terms of supported features, constantly increasing the sophistication of GPUs is effective for improving visual quality, but does not address the problem of driving simultaneous displays at increased resolutions. Also, multi-GPU solutions have their own limits—they do not properly distribute and scale real-time rendering beyond the scope of one PC.

Industries outside games that are interested in projecting to highly complex display systems now need to look beyond game technology to find solutions. Reliable solutions for distributing pre-rendered content, such as video playback applications, have been on the market for a very long time. However, the same cannot be said for the distribution of real-time content; no single solution exists that offers an effective mechanism to scale real-time and generative content to arbitrary sizes.

Display setups with multiple screens of various shapes and curvatures, and with various file formats and resolutions, are just a few of the complexities faced by those attempting scaled real-time displays. In addition, content providers for real-time displays are constantly pushing the envelope of sophistication and realism in visuals, which means hardware needs to be updated frequently to keep up with the demand.

Image courtesy of Reynaers Aluminium

Types of display systems

Before we dive into the needs for scaled rendering, let's review some of the hardware and usage involved.

Display solutions for either playback or real-time content take two forms:

- Monitors or LED screens, where image data is transferred typically via a cable to one or many processors driving the pixels
- Projection, where imagery is reproduced on an arbitrary surface via multiple film projectors

The topology of the display system can fall into any of a number of categories:

- Matrix of LED screens
- Very large LED screen, curved or flat
- Flat-screen projection
- Projection on dome or curved surface
- Cave automatic virtual environment (CAVE) projection-based multi-sided immersive environment
- Complex displays utilizing two or more of the above

These displays might also have a requirement for stereoscopic vision, or for synchronization and tracking capabilities to accurately represent the user point of view in 3D space.

Industries that use high-end complex displays include:

- Virtual production - In-camera VFX with projection or LED imagery in place of green screen
- Architecture and manufacturing - CAVE or powerwall[1] for design review
- Simulation-based training - Tilted walls or curved screens to immerse the participant in the environment
- Entertainment - Planetariums and theme park rides with projection domes
- Live events and permanent installations - Displays using projection or LED screens with extraordinary sizes and resolutions that require a large number of servers to drive content simulation and playback

Image courtesy of Wolf + Rothstein

[1]A *powerwall* is a large viewing surface, either LED or projective, that displays imagery at a high enough resolution for the viewer to see detail even when close to the screen. Powerwalls are typically used for collaborative applications such as architectural or engineering design review, or for installations such as museums where viewers can get close to the display. Due to the amount of data required for such high resolution imagery, a powerwall system is driven by multiple PCs. A projective powerwall requires multiple projectors, while an LED powerwall can consist of one or many LED screens.

Use cases

For context, we show here some uses cases for scaled display of real-time content. Epic assisted these partners in their endeavors alongside PixelaLabs, a team of rendering, VR, and CAVE specialists that provide nDisplay integration services for large-scale or customized projects.

Live event / Dome projection

In 2018, hip-hop performer Childish Gambino performed his Pharos show, which featured real-time projections inside an enormous dome throughout the concert. The team rendered a 5.4K by 5.4K image on five machines, then split into the image into a fisheye and sent it to 12 projectors.


Image courtesy of Wolf + Rothstein


Image courtesy of Lune Rouge Entertainment

Live event / Complex display

PY1 is a traveling 81-foot pyramid-shaped venue designed to serve a variety of entertainment purposes. The projection system uses 32 projectors, and rental of the venue includes lasers, kinetic stage elements, and special effects.

CAVE / Architectural visualization
Simulation-based training

Reynaers Aluminium installed AVALON, a five-sided CAVE, at their headquarters in Duffel, Belgium to display architectural design and window installation training. Viewers wear active VR glasses to see the imagery in 3D with a natural field of view. The AVALON system uses 25 projectors and 14 workstations.


Image courtesy of Reynaers Aluminium

Virtual production / Flat LED

Lux Machina constructed an LED volume consisting of four LED panels (three walls and a ceiling) to surround actors and props, and to light them and provide reflections. The view through the physical camera shows real-world elements seamlessly integrated into the real-time CG environment.

Technical considerations

To begin our inspection of the challenges of distributing real-time content, let's take a closer look at the various technical considerations involved.

Display mechanisms

Large-scale real-time displays show multiple sections of a single frame at a time. How these multiple sections are stitched to form one coherent image depends on the size and shape of the display, and the underlying display technology.

Projection screens

With a projective setup, images are projected onto surfaces with projectors. The frame section from each projector overlaps adjacent sections for smooth blending, usually by around 15-20% of the section size. Projection screens for real-time display can take any of several forms:

- **Planar or curved** - Projectors are stacked horizontally, vertically, or in both directions for larger setups or when brightness requirements are higher. Content overlaps and is blended in the overlap zone.
- **Spherical or pyramidal** - A complex array of projectors covers the entire surface with often significant overlap for increased brightness.
- **Arbitrary shapes** - Virtually any shape works as a projection surface, provided the image is visible at sufficient brightness.

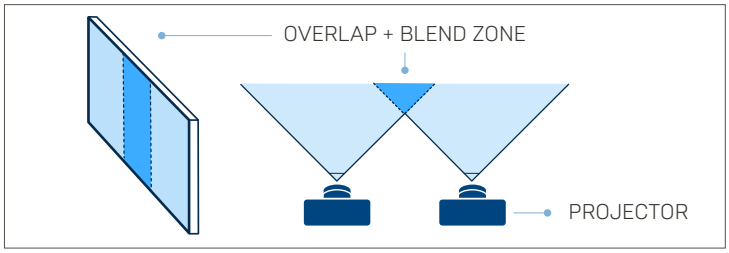

Figure 1: Planar surface and projector setup

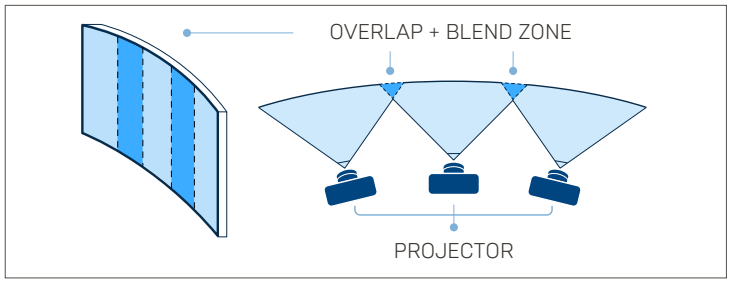

Figure 2: Curved surface and projection setup

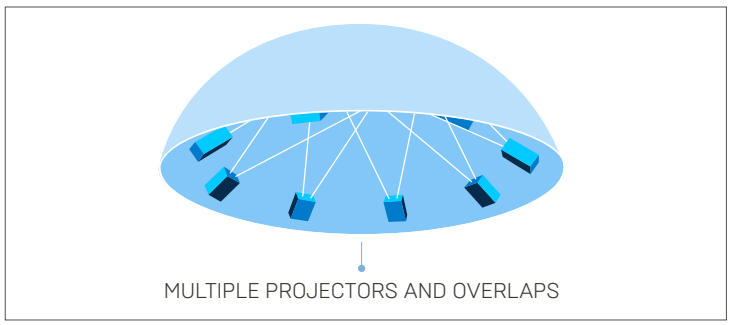

Figure 3: Spherical or dome surface and projector setup

Figure 4: Examples of flat, curved, and spherical projection screen setups. [Images courtesy of Scalable Display Technologies]

Figure 5: Example of curved LED screen setup. [Image courtesy of Moment Factory]

LED screens

LED screens were, until recently, always flat, and any curved shape was created by placing flat screens at a slight angle to one another. Nowadays, LED screens can be designed to almost any form, shape, resolution, or pixel pitch. Flat LED screens can also be arranged in complex patterns to form three-dimensional displays.

Because the image data for an LED screen comes via a cable rather than a projector, the seams between image portions can be precisely lined up, and thus overlap/blending between portions is not necessary.

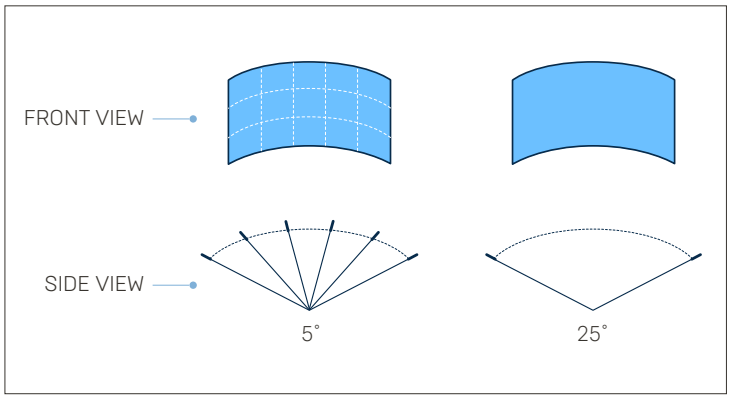

Figure 6: Curved LED display from flat screens (left) and single curved panel (right)

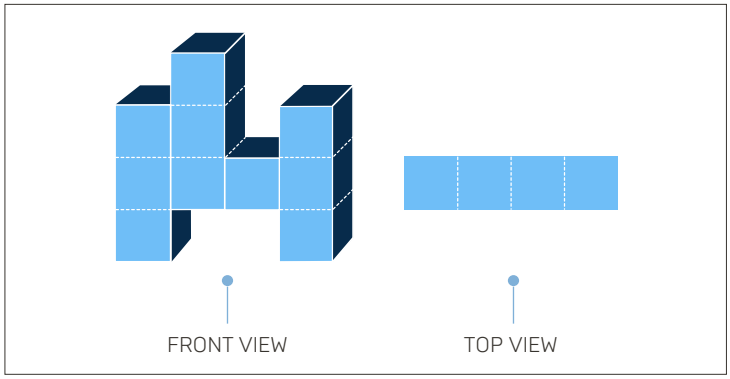

Figure 7: 3D display built from multiple flat LED panels

Synchronization

Naturally, synchronization is a vital part of displaying multiple sections of real-time content over a large display. All systems within the render/display ecosystem must adhere to strict timing, measured in milliseconds, to produce the illusion of a seamless display.

Multi-display setups often require syncing capabilities at both the software and hardware levels. Not only should the generated content be ready at the same time on all PCs using the same timing information for simulation, but the display swap (the changing out of the current image for the next image in the video card buffer) needs to also happen at the correct time to prevent "tearing" artifacts in the display.

For VR and other types of stereoscopic displays, the issue of synchronization applies doubly since the two different frames, one for each eye, must coordinate perfectly.

Here we will discuss the aspects of synchronization that are most relevant to nDisplay. You can learn more about the implementation of these features in the Synchronization in nDisplay topic in the Unreal Engine documentation.

Determinism

There are two types of approaches for managing synchronization:

- **Deterministic:** Each server (PC, rendering node) is set up in such a way that the output is always predictable given a particular set of inputs, which means the only information the server needs to synchronize with other machines in the system is an accurate time, and input/output information for each individual machine.
- **Non-deterministic:** To ensure synchronization, the system forces replication of the transform matrices and other relevant characteristics of all actors or objects in a scene and reproduces them throughout the system.

Each approach has pros and cons. The main advantage of a deterministic system is project simplicity, and the data

bandwidth saved by not sharing transform data for each object at every frame. The downside is that if one system diverges, the divergence will have unknown issues over time. Rendering uniformity could be severely compromised, leading to visual discontinuity and artifacts.

Hardware sync and genlock

While the nDisplay primary PC ensures that all cluster node PCs in the cluster are informed of timing information from a gameplay perspective (for example, which frame to render), specialized hardware sync cards and compatible professional graphics cards are necessary to synchronize the display of those rendered frames at exactly the same time on the physical display devices.

In broadcast applications, for example, it is common to synchronize many devices such as cameras, monitors, and other displays so they all switch and capture the next frame at precisely the same time. In this industry, the use of genlock is widely used and adopted.

Typically, the setup is composed of a hardware generator that sends the clock to the hardware requiring synchronization. In the case of PCs used for real-time rendering, professional graphics cards such as those in the NVIDIA Quadro line support this technology alongside the NVIDIA Quadro Sync II card, which will lock to the received timing signal or pulse.

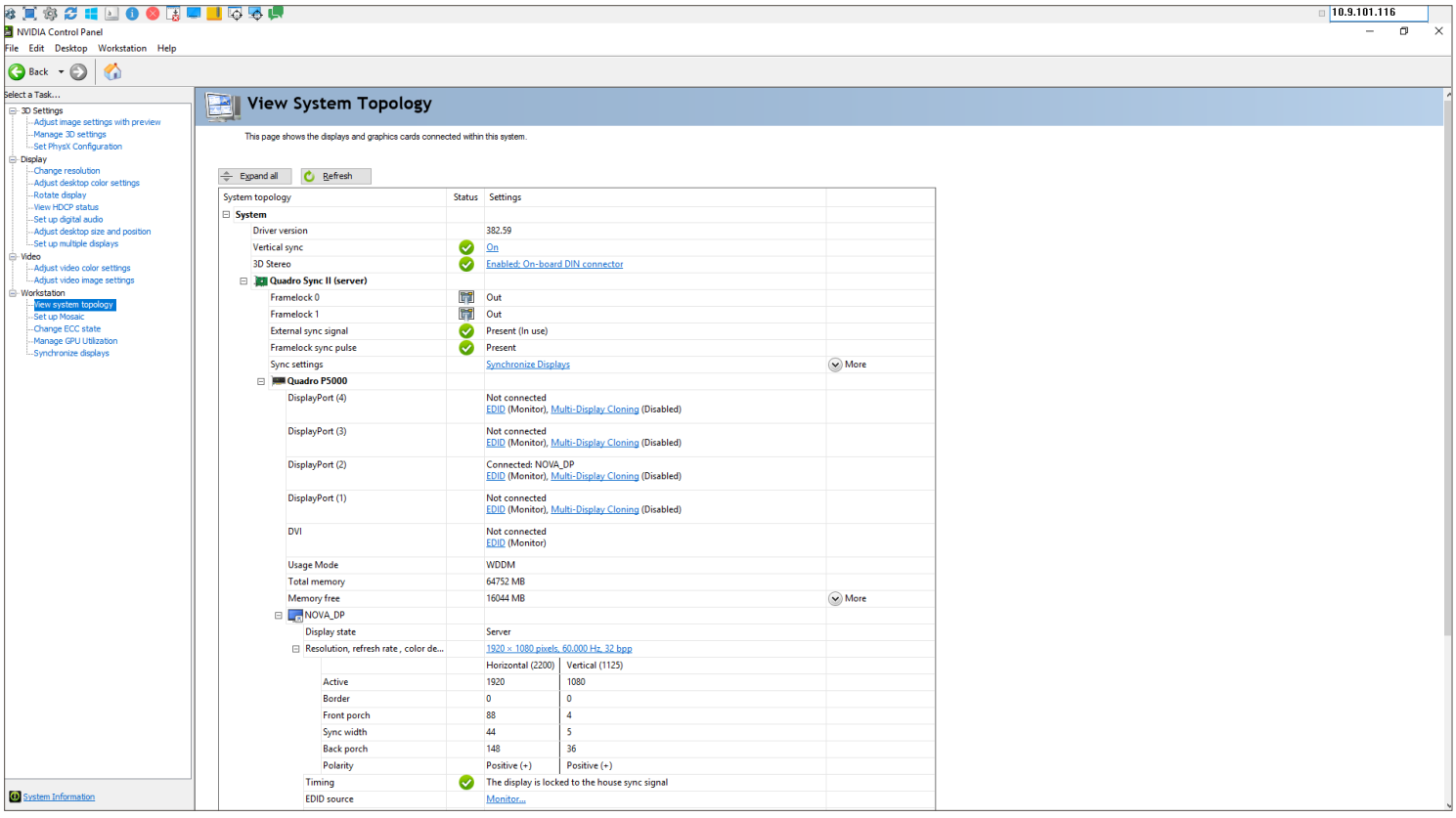

Figure 8: Screenshot of NVIDIA Control Panel - View System Topology for genlock

Daisy chain versus direct genlock

Daisy chaining is a signal-locking technique that is used alongside direct genlock, where the master clock is sent to a single PC or device—in our case, the Primary PC. Separate cables then propagate the signal to all other PCs.

Previous experience with nDisplay suggests that direct genlock, where each PC receives the clock directly from the master source, is simpler and more effective than daisy chaining. However, a new hardware approach based on daisy-chaining, NVIDIA Swap Sync/Lock, was released as part of Unreal Engine 4.25 to provide an alternative solution to signal locking that may be more reliable and cost-effective.

Synchronization testing

Testing synchronization for a scaled display can be tricky because desynchronization can result from any number of issues, including:

- Wrong frame is simulated due to incorrect timestamp
- Display device timing is off

To test sync, we use a simple test project displaying a single object moving quickly across the entire display surface. When systems are properly in sync, the object retains its form as it passes across boundaries. Otherwise, the display will show artifacts at shared edges.

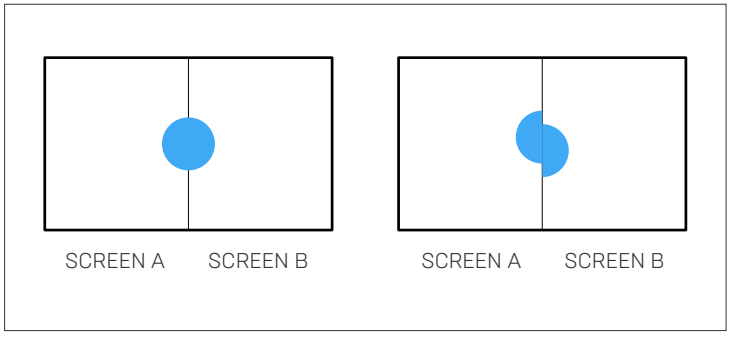

Figure 9: Synchronization testing on a simple scene. Setup at left has correct synchronization, setup at right does not.

Post effects

Post-production effects such as bloom, lens flare, and motion blur are computed in screen space, meaning they can be applied only after the entire frame is rendered. This is due to the fact that such effects are usually not isolated to one section of the image; they need information about the pixels in neighboring sections to render properly.

For example, the effect of a single bloom often spreads widely throughout an image. In a distributed rendering system using the deterministic approach, a bloom that originates from a spot in one portion of the image will not "reach" neighboring sections, and the blend between the bloom area and non-bloom area will not be smooth.

For scaled displays, such effects should be disabled in order to prevent visual artifacts in blend areas. It is possible to have these types of effects, but they must be managed very carefully at display boundaries, usually using costly and advanced techniques.

Existing technology

As part of the development strategy for new features, Epic Games is constantly evaluating existing tools that could add functionality to Unreal Engine (UE4). After much research, we found the following technologies to help us achieve our goals for scaled displays.

MPCDI

The MPCDI (Multiple Projection Common Data Interchange) standard was developed by VESA's Multi-Projector Automatic Calibration (MPAC) Task Group. This is a standard data format for projection calibration systems to communicate with devices in a multi-display configuration. The standard provides a way for multi-projector systems to generate the data needed to combine individual display components into a single, seamless image by a variety of devices. Any new hardware introduced into a system can be easily integrated with the standard.

MPCDI is used throughout the industry by content producers and vendors such as:

- Scalable Display Technologies
- VIOSO
- Dataton Watchout
- 7thSense Design

Scalable Display EasyBlend

Scalable Display Technologies is a company that focuses on software and SDKs for complex projection systems. Their SDK is designed to provide a solution for large displays of a single image through warping and blending. Since Scalable Display Technologies already had the EasyBlend solution in place to handle warping and blending of large images, we chose to integrate it with Unreal Engine to achieve our goals.

System specification

After considering the most likely or common needs for scaled real-time content, and also reviewing available technology, we identified the following as requirements for a system to work with Unreal Engine. The system needed to be able to:

- Deploy and launch multiple instances of Unreal Engine across an array of computers in a network
- Ensure perfectly synchronized content restitution on all computers involved
- Enable active/passive stereoscopic vision
- Manage and distribute various sources of inputs such as VR tracking systems
- Accept various configurations of displays in terms of size, spatial orientation, and resolution

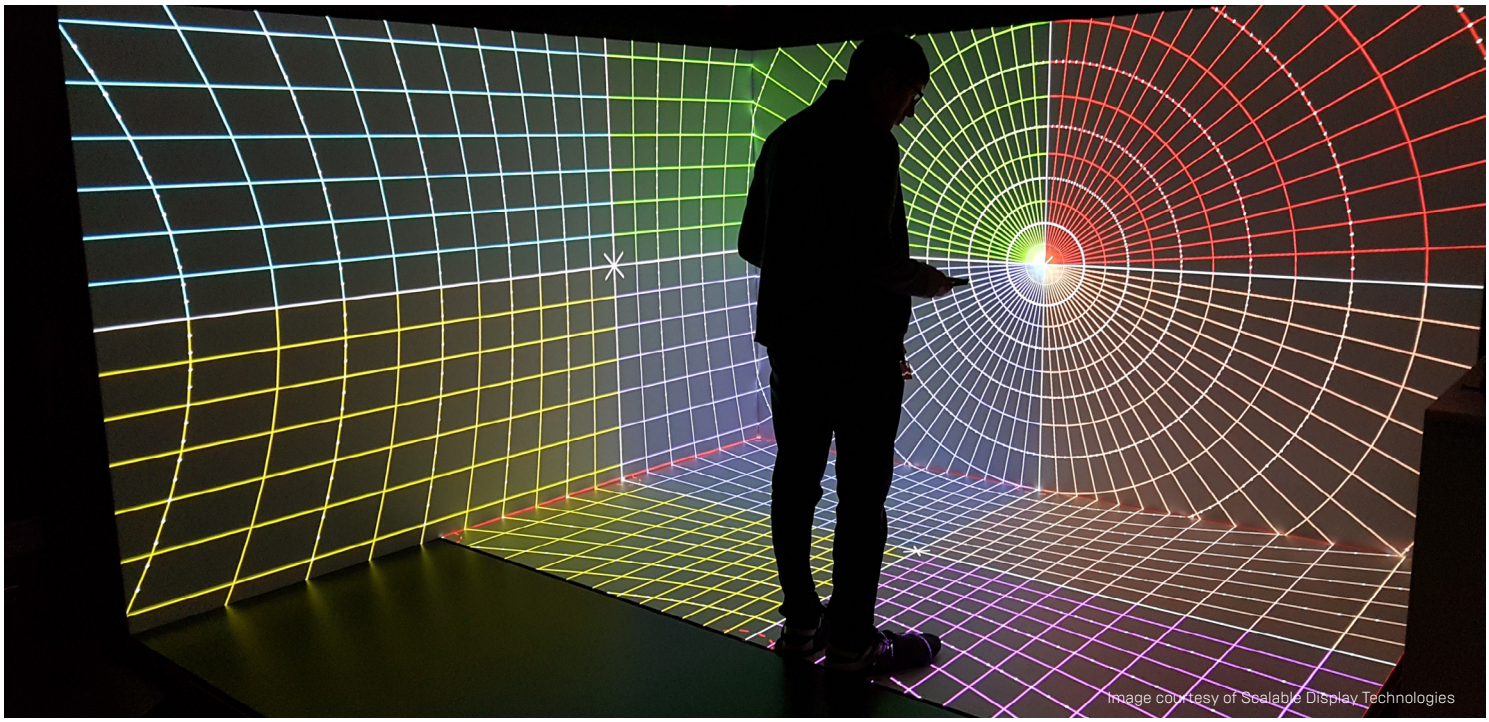

Image courtesy of Scalable Display Technologies

nDisplay solution

To meet these requirements, Epic Games developed the nDisplay system. nDisplay distributes the rendering of Unreal Engine content across a network of computers to generate images to as many displays as required with proper frame/time synchronization, correct viewing frustum based on the topology of the screens in world space, and deterministic content that is identical across the visualization system.

In its essence, nDisplay technology extends Unreal Engine by distributing the rendering of a camera view over an arbitrary number of machines and then displaying the rendered images on an arbitrary number of display mechanisms.

After much consideration, we decided that the best way to implement nDisplay with Unreal Engine was to have the cluster node automatically attach itself to the active camera location in the Unreal Engine project. The view from the Unreal Engine camera is what is extended, rendered, and distributed based on the nDisplay settings.

nDisplay does the following:

- Synchronizes actors across Unreal Engine Instances (cameras, animations, particles, etc.)
- Acts as a listener for keys, axis, and positioning
- Acts as VRPN server for tracking devices such as ART and Vicon
- Supports DirectX 11 and DirectX 12 graphics libraries in both mono and stereoscopic modes (frame sequential-quad buffer, side by side, top / bottom)
- Supports NVIDIA Quadro Sync synchronization for frame consistency, flexibility, and scalability
- Provides asymmetric frustums configuration for stereoscopic systems

Structure

The nDisplay toolset consists of a plugin and a set of configuration files and applications for Unreal Engine. It includes the following components:

- nDisplay plugin - Used during runtime to provide the network interface between instances, optional actor replication, and input management system; configures rendering subsystem to display node topology
- nDisplay configuration file - Describes the topology of the display system and overall centralized location for project settings
- nDisplayLauncher and nDisplayListener - Applications for launching and controlling “n” instances of Unreal Engine across different computers on a network, each connected to one or many displays

Integration with Unreal Engine features

Deterministic and non-deterministic features

Earlier in this paper, we discussed the pros and cons of a deterministic system, where synchronization is simplified by each node not sharing all frame information with other nodes.

In the case of Unreal Engine, we lean towards fully deterministic systems in terms of gameplay, physics, and rendering features. However, some subfeatures are currently not fully deterministic. The chart below shows where we stand in terms of deterministic features within Unreal Engine.

Feature	Deterministic?	Notes
Simple physics	To a limited extent	Collision volumes, capsules, planes, boxes, and spheres exhibit incoherent behavior using all physics solvers. Replication is required for accuracy.
Complex physics • Rigid bodies • Soft bodies • Cloth • Vehicles and joints • Skeletons	To a limited extent	Random noise introduced deep in the current PhysX solver prevents the system from maintaining synchronization over time. Replication is required for accuracy.
Niagara particles	Yes	We have not yet experienced time divergence using this system.
Sequencer	Yes	Yes, deterministic in nature, functions as expected.
Blueprint gameplay logic	Yes	Most Blueprint logic will function as expected unless the Blueprint contains functionality known to not be deterministic, such as using non-deterministic devices or using not-yet-deterministic functions.
Blueprint randomized logic	No	By nature, randomization features across Unreal Engine prevent logic from being fully deterministic. Note that some particle systems configured in Blueprint are randomized and should be used and reviewed with care.

Table 1: Determinism of Unreal Engine features

Our future plans include rewriting the Unreal physics engine to fully support deterministic behavior. In the meantime, we do support replication capabilities that enforce visual coherence when needed, at the cost of extra bandwidth and some project customization.

Asymmetric frustums

In the traditional use of a camera, the camera is centered in front of the view, providing a symmetric frustum.

A distributed system requires the definition of camera locations and custom view frustums which are assigned to various PCs for rendering. Because one UE4 camera supplies the view for multiple screens or projections, the camera frustum must understandably be split into more than one frustum, with each frustum providing the imagery for a specific projector or portion of an LED screen. These frustums are, by nature, asymmetric.

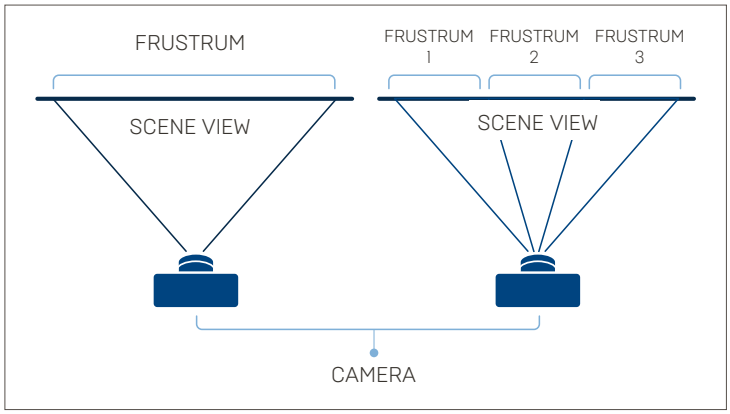

Figure 10: Symmetric view frustum (left) and asymmetric view frustums (right)

Note that the perspective in the rendered image does not change when the frustum is split. The splitting of the frustum is simply to facilitate distributed rendering of the camera view, which is often off-center with nDisplay systems.

Post-process VFX

Because of the potential issues with continuity across screen junctions or overlap/blend areas for projectors, the suggested workflow is to disable screen-space post processes that have been identified as potential causes of rendering artifacts. However, if a project really needs these effects, one possible solution is to render extra pixels beyond the actual viewing frustum. This approach, called *overscan*, allows some of these effects to be applied at the cost of extra render time. You can develop this approach by extending nDisplay.

The following is a list of post-process VFX that should be disabled or used with caution, as they pose a risk of tearing or other continuity issues:

- Bloom
- Lens flare
- Automatic eye adaptation
- Motion blur
- Ambient occlusion
- Anti-aliasing (although very subtle, some techniques might show a difference)
- Screen-space reflections
- Vignetting
- Chromatic aberration

When deciding whether to include screen-space effects in a distributed display, you will need to balance the importance of the effect to the user experience with the extra care involved in making the effect work (and the possibility that the effect won't work properly and will end up detracting from the experience rather than enhancing it). The many facets involved in making such a decision for any particular project are beyond the scope of this paper.

MPCDI

Support for the MPCDI standard enables nDisplay to read and store data describing a complex projector system in a standardized and formalized fashion, so that we can easily communicate and interface with various other tools from within the industry.

Because the MPCDI implementation is new, its usability and UX are still somewhat lacking. To get around these limitations, we are working on a solution for previewing the MPCDI file data within the Unreal Editor and at runtime.

Currently, users are able to generate procedural meshes of physical displays based on mesh data generated from the MPCDI file.

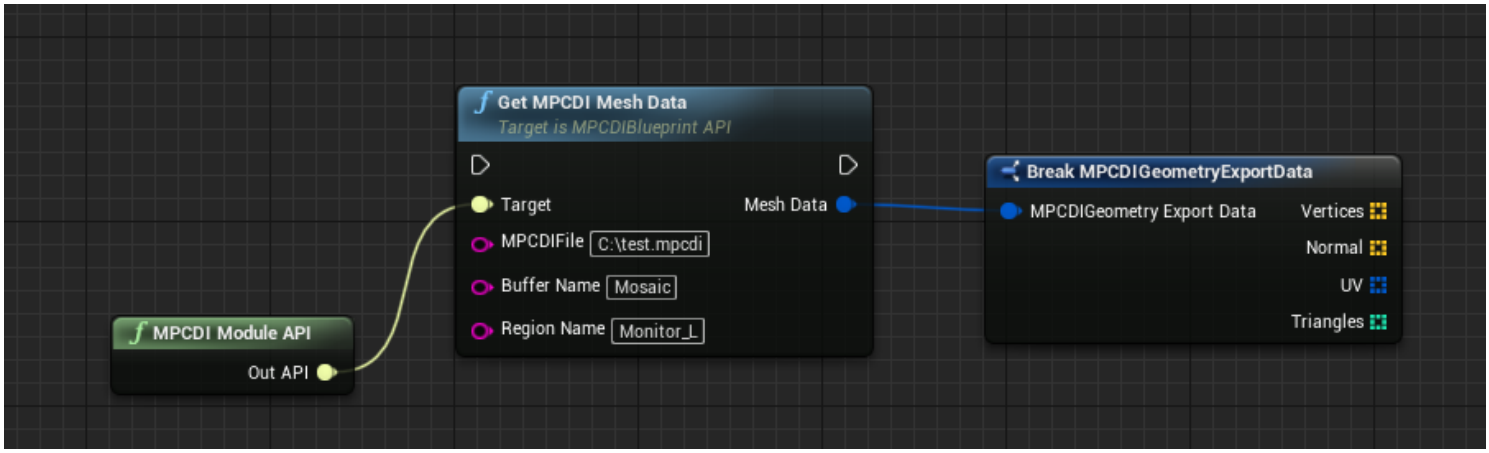

Figure 11: Example of MPCDI setup in Blueprint

At the time of the writing, no commercial tools are available for implementing MPCDI, so the generation and manipulation of MPCDI configuration files are the responsibility of the underlying software companies. In the future, we may introduce a display or editing tool for the MPCDI configuration file as part of nDisplay, but this is not part of our current implementation.

Scalable Display - EasyBlend integration

nDisplay supports warp and blend through the integration of industry-standard middleware Scalable SDK and EasyBlend for all supported modes, native warp and blend with MPCDI, and custom implementations.

We implemented the integration of EasyBlend to provide a seamless experience in configuring a complex projective system. Once calibration is completed using the third-party tool or software, the user only needs to specify a few parameters in the nDisplay configuration file to get it running.

Limitations

The following manual steps are required to set up and use nDisplay:

- **Configuration:** Manually set up your configuration file by defining your display topology, projection policy, viewports, render node PCs, tracking devices, and any other components of the system.
- **Project adaptation:** With recent updates to nDisplay, only slight adaptations to your project are required to ensure it isn't using incompatible or non-deterministic features. Essentially, everything that is linear, animated, and does not rely on complex physics, randomized functions, or screen-space VFX will work as is. More complex projects might require replication of some actors in order to be properly synced.
- **Deployment:** Configure and copy your project files and assets to the destination render PCs, either manually or by using custom tools. Then use our provided third-party tools to launch your nDisplay project remotely.

Here are some known physical limitations in nDisplay, at the time of writing:

- nDisplay currently runs on Windows 7 and higher (DirectX 11 and DirectX 12).
- Quad buffer (active) stereo feature is supported only on Windows 8.1 and higher.
- OpenGL support is deprecated.
- nDisplay does not currently have Linux support.
- Support for framelock and genlock is available only for professional-grade graphics cards such as NVIDIA Quadro.
- Auto exposure, bloom, planar reflections, and some shader effects are not supported by nDisplay. You can still use them, but you will likely see visual artifacts or discontinuities in your displayed content.
- 2D UMG Interface features (Hover, Click, and Viewport Alignment) are not supported. Alternative methods through OSC, REST, or other available remote control protocols can be used to control an nDisplay system.

Note that nDisplay scales the GPU side of rendering, which is the main bottleneck for real-time content, but it doesn't scale CPU-based operations like gameplay and physics. All cluster PCs will still have to process all these CPU-based operations individually. In other words, because nDisplay is strictly a rendering distribution system, it won't provide any acceleration from a CPU standpoint.

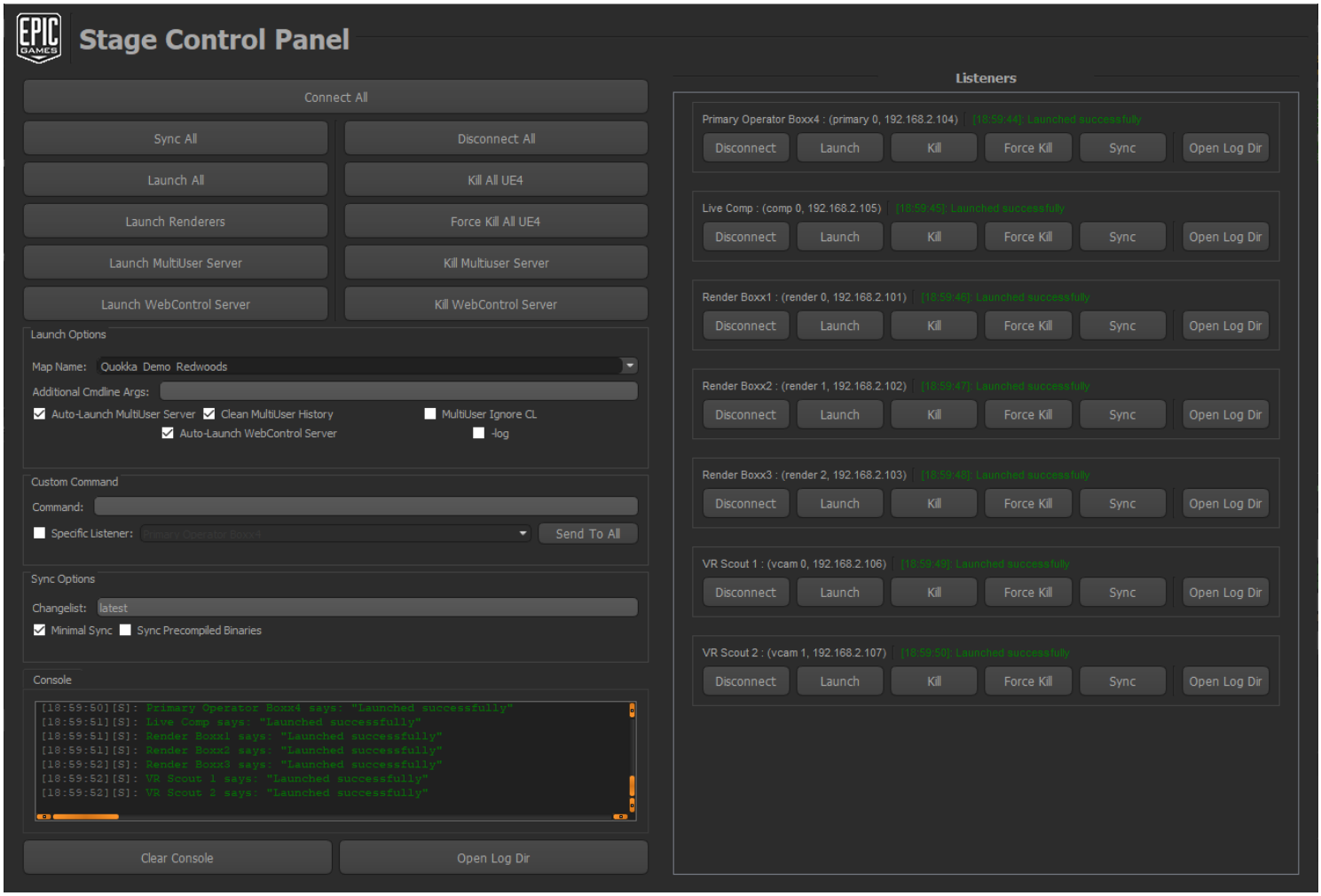

Figure 12: Example of custom deployment tool

Deployment

At the time of writing, nDisplay does not offer any automated content or project deployment tools. This is because most users often have very large and complex projects and specific network constraints such as firewall policies, and often prefer their own ways of managing and deploying/copying content.

In the future, however, we expect to include a minimal set of quick deployment features that you can use to test nDisplay, further reducing the manual steps required to scale the rendering of Unreal Engine-based content.

In the meantime, internal teams have come up with tools like the one shown in Figure 12 from which we will gain inspiration to drive future versions of the current Deployment Tool.

nDisplay workflow

The following diagrams show how nDisplay works with a network and display devices.

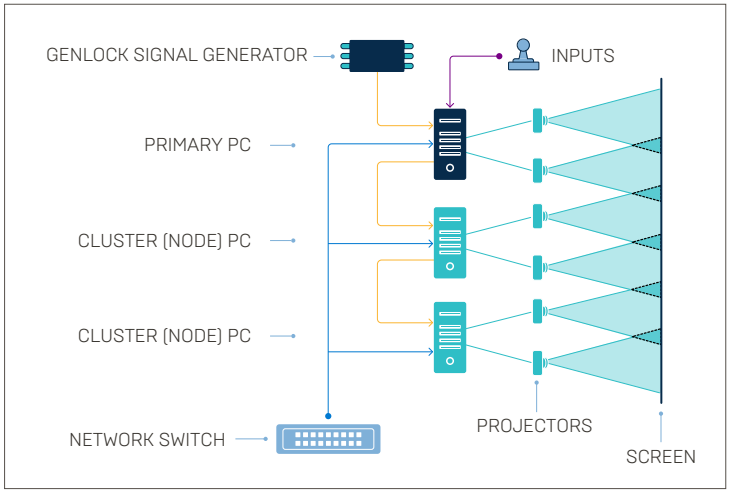

Figure 13: Network setup for projective display

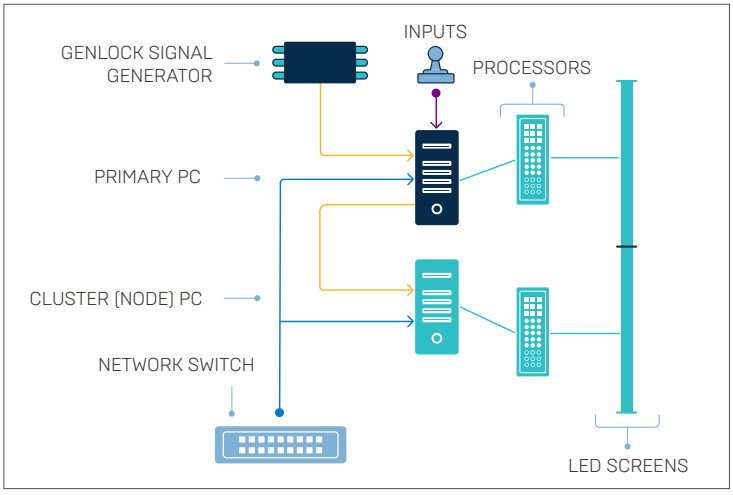

Figure 14: Network setup for LED display

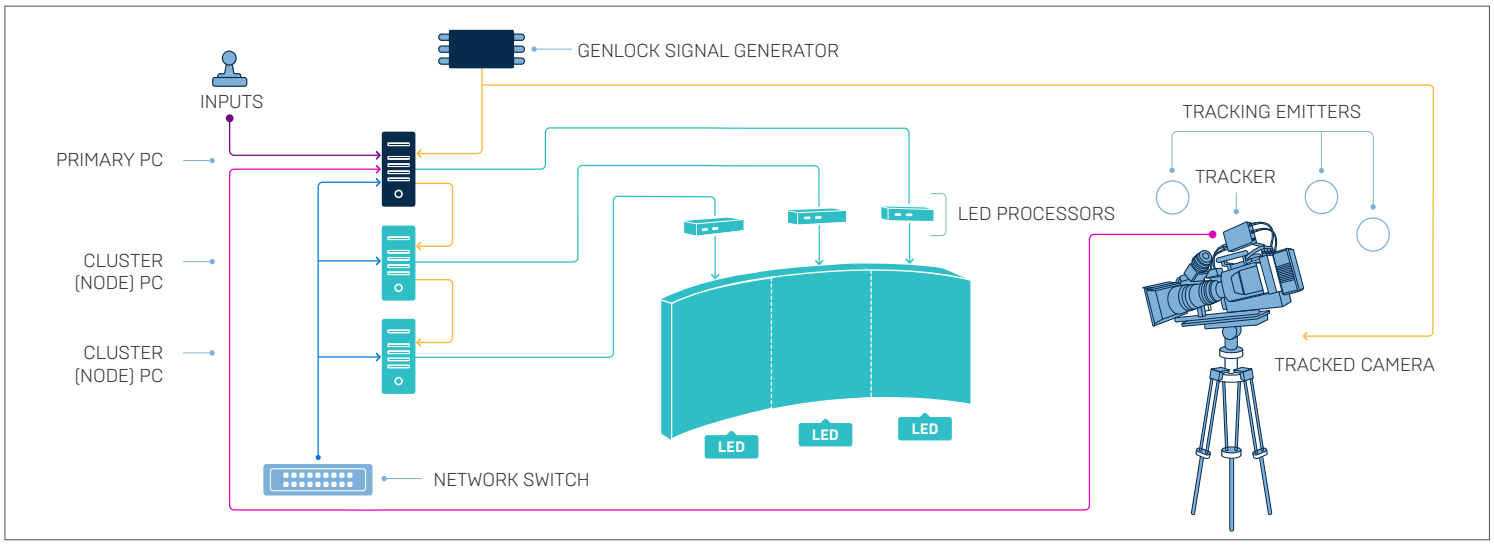

Figure 15: Network setup for projective system with tracked camera

The nDisplay plugin can be enabled for an existing project, or can be enabled automatically by creating a project with the nDisplay template. See the nDisplay Quick Start documentation for details.

Primary PC

The main computer acting as nDisplay conductor. Centralized location for:

- Managing and dispatching inputs as well as camera tracking data and custom cluster events to the other cluster (node) PCs in the nDisplay cluster network in a synchronous manner
- Ensuring all PCs receive and acknowledge the same inputs and data at the same time
- Managing timing information across cluster of PCs
- Managing and distributing optional actor and data replication to the other PCs

Cluster (node) PCs

- Effectively execute identical gameplay and physics simulation to the Primary PC
- Render extra camera frustums in synchronous manner with the Primary PC
- Frame-locked and genlocked with all cluster PCs

Configuration file

The configuration file is at the root of nDisplay functionality. In essence, it describes your hardware setup in terms of PCs and display mechanisms as well as the relationship between them. While the setup of the nDisplay configuration file is described in detail in our nDisplay Configuration File Reference documentation, in this section we'll go over a few concepts and terms specific to nDisplay.

Windows, viewports, and screens

To configure an nDisplay setup, it's important to understand a few key terms with very specific meanings in the context of nDisplay.

Window - The portion of the entire frame that will be rendered/displayed by a single node in the cluster (usually a single PC).

Viewport - A rectangular portion of a window. For example, the image in a window might be made up of four viewports, each of which is rendered separately and then assembled in the window. The nDisplay configuration specifies how many viewports make up any one window, and gives the viewports' positions within the window. Note that it is perfectly valid to set up just one viewport for a window.

Screen - The physical positioning, size, and orientation of the displays in the real world. With projection, the screen is a rectangle which determines the camera frustum.

Camera - An offset to the camera inside the Unreal Engine scene. This offset can be useful for adjusting the imagery by a small amount to improve the viewer's experience.

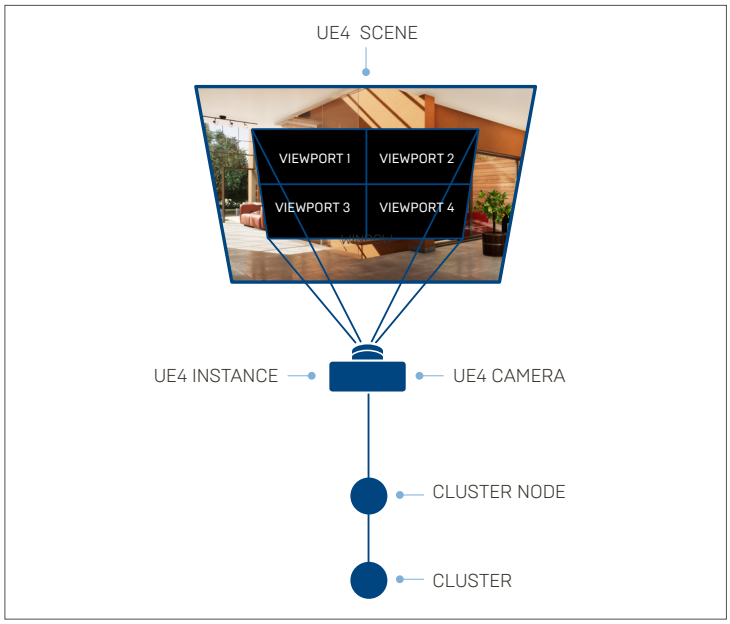

Figure 16: Relationship between window, viewports, instance, camera, and cluster node in nDisplay setup with a single cluster node.

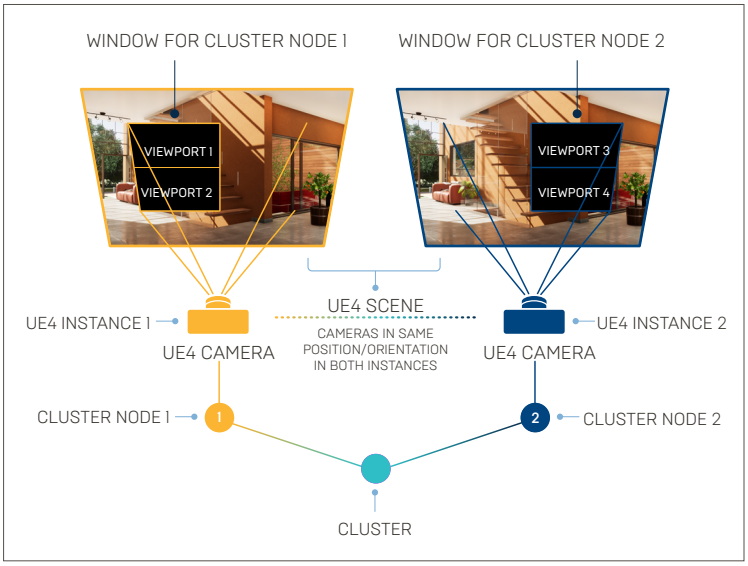

Figure 17: Relationship between windows, viewports, instances, camera, and cluster nodes in nDisplay setup with more than one cluster node.

The following rules help to illustrate the relationships between these elements:

- A cluster has one or more cluster nodes.
- A cluster node corresponds to a single UE4 camera, but a single UE4 camera may have many cluster nodes hooked to it.
- A cluster node has only one window, which is some portion or all of the UE4 camera view.
- A window has one or more viewports.
- Each viewport has one projection policy and shows a portion of the view (or the entire view) from one of the available cameras in the nDisplay configuration file. The same projection policy can be used by multiple viewports.
- Viewports cannot overlap.
- A camera can be re-used by different viewports.

Clusters and cluster nodes

Cluster nodes are used to describe the nDisplay PC network configurations, designate which machine will be the primary server, and specify which windows are assigned to each PC. The cluster is built and hooked to the active camera in the Unreal Engine scene, where the camera view is what is being rendered and distributed.

nDisplay essentially adds additional viewpoints to your current Unreal Engine camera. Think of additional eyes or, more precisely, virtual viewports looking at your scene in arbitrary locations, but attached to the master camera path. If you move the camera in your Unreal Engine scene forward by five units, then the whole nDisplay cluster moves in the same direction by five units.

A cluster node is associated with a single UE4 application instance. Usually each application instance runs on its own machine, but you may use multiple instances on the same PC with a separate cluster node associated with each instance.

The cluster nodes make up the cluster. The most common setup, for a single contiguous display of a single UE4 camera view, needs only one cluster. A less common setup consists of two or more screens which show different UE4 scenes (or different camera angles within a common scene). In such a case, you would need to set up a separate cluster for each UE4 scene.

Projection policy

A projection policy is an abstraction that specifies where to send a projection's input data and how to compute the output. This means that each policy might have its own properties that it knows how to interpret and utilize.

nDisplay supports several policies. The following are the most commonly used:

- **Simple** - Standard policy used to render on regular displays.
- **EasyBlend** - Integration of EasyBlend calibration data by Scalable SDK, enabling warp/blend/keystoning features. Required to display on non-planar and complex display surfaces such as curved or dome-shaped surfaces using multi-projectors.
- **MPCDI** - Integration of the MPCDI standard, used for complex projects relying on this industry protocol.

Config file examples

Complete EasyBlend policy config file example

```
[info] version="23"

[cluster_node] id="node_left"  addr="10.1.100.2"  window="wnd_left"   master="true"
[cluster_node] id="node_right" addr="10.1.100.3"  window="wnd_right"

[window] id="wnd_left"  fullscreen="true" viewports="vp_1,vp_2"
[window] id="wnd_right" fullscreen="true" viewports="vp_3,vp_4"

[projection] id=proj_easyblend_1 type="easyblend" file="C:\Program Files\Scalable Display\DEI\LocalCalibration\ScalableData.pol"   origin=easyblend_origin scale=1
[projection] id=proj_easyblend_2 type="easyblend" file="C:\Program Files\Scalable Display\DEI\LocalCalibration\ScalableData.pol_1" origin=easyblend_origin scale=1
[projection] id=proj_easyblend_3 type="easyblend" file="C:\Program Files\Scalable Display\DEI\LocalCalibration\ScalableData.pol"   origin=easyblend_origin scale=1
[projection] id=proj_easyblend_4 type="easyblend" file="C:\Program Files\Scalable Display\DEI\LocalCalibration\ScalableData.pol_1" origin=easyblend_origin scale=1

[viewport] id=vp_1 x=0 y=0    width=2560 height=1600 projection=proj_easyblend_3
[viewport] id=vp_2 x=2560 y=0 width=2560 height=1600 projection=proj_easyblend_4
[viewport] id=vp_3 x=0 y=0    width=2560 height=1600 projection=proj_easyblend_1
[viewport] id=vp_4 x=2560 y=0 width=2560 height=1600 projection=proj_easyblend_2

[camera] id=camera_static loc="X=0,Y=0,Z=0.0"

[scene_node] id=cave_origin  loc="X=0,Y=0,Z=0"   rot="P=0,Y=0,R=0"
[scene_node] id=wand         loc="X=0,Y=0,Z=1"
[scene_node] id=easyblend_origin loc="X=0,Y=0,Z=0" rot="P=0,Y=0,R=0"

[general] swap_sync_policy=1   ue4_input_sync_policy=1
[network] cln_conn_tries_amount=10 cln_conn_retry_delay=1000 game_start_timeout=30000 barrier_wait_timeout=5000
[custom] SampleArg1=SampleVal1 SampleArg2=SampleVal2
```

Complete simple policy dual monitor config file example

```
[info] version="23"

[cluster_node] id="node_left"  addr="127.0.0.1"  window="wnd_left"   master="true"
[cluster_node] id="node_right" addr="127.0.0.1"  window="wnd_right"

[window] id="wnd_left"  fullscreen="true" viewports="vp_left"  WinX="0"    WinY="0" ResX="2560" ResY="1440"
[window] id="wnd_right" fullscreen="true" viewports="vp_right" WinX="2560" WinY="0" ResX="2560" ResY="1440"

[projection] id="proj_left"  type="simple" screen="scr_left"
[projection] id="proj_right" type="simple" screen="scr_right"

[screen] id="scr_left"  loc="X=1.5,Y=-.8889,Z=0" rot="P=0,Y=0,R=0" size="X=1.7778,Y=1.0"
[screen] id="scr_right" loc="X=1.5,Y=.8889,Z=0" rot="P=0,Y=0,R=0" size="X=1.7778,Y=1.0"

[viewport] id="vp_left"  x="0" y="0" width="2560" height="1440" projection="proj_left" camera="camera_left"
[viewport] id="vp_right" x="0" y="0" width="2560" height="1440" projection="proj_right" camera="camera_right"

[camera] id=camera_left loc="X=0,Y=0,Z=0.0"
[camera] id=camera_right loc="X=0,Y=0,Z=0.0"

[scene_node] id=cave_origin  loc="X=0,Y=0,Z=0"   rot="P=0,Y=0,R=0"
[scene_node] id=wand         loc="X=0,Y=0,Z=1"
[scene_node] id=proj_origin loc="X=0,Y=0,Z=0" rot="P=0,Y=0,R=0"

[general] swap_sync_policy=1   ue4_input_sync_policy=1
[network] cln_conn_tries_amount=30000 cln_conn_retry_delay=500 game_start_timeout=30000000 barrier_wait_timeout=50000000
[custom] SampleArg1=SampleVal1 SampleArg2=SampleVal2
```

nDisplayLauncher and nDisplayListener

These applications are shipped with the nDisplay plugin in order to facilitate the deployment of a project across an array of PCs.

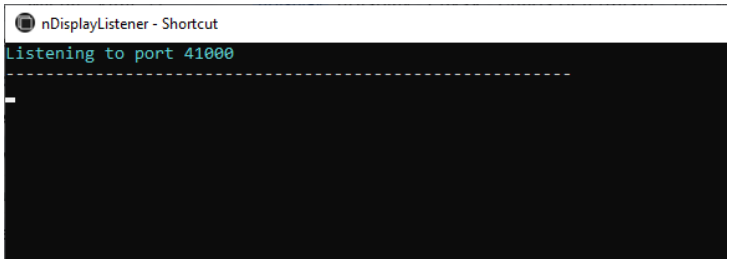

Figure 18: Screenshot of nDisplayListener software

Listener

The nDisplayListener is a minimalist application that resides on the host PCs (primary and all cluster node PCs) and can receive various remote commands, for example to launch an existing project using a path and argument list, or to terminate an existing project.

Launcher

The Launcher simultaneously launches multiple projects on a list of available PCs that are running the Listener in the background. The Launcher can be run from any PC or laptop on the local network.

To run the Launcher, specify the following:

- Application path
- Configuration file describing the cluster network, display topology, and other required settings
- Optional settings such as stereoscopic settings, project variables, or command line arguments

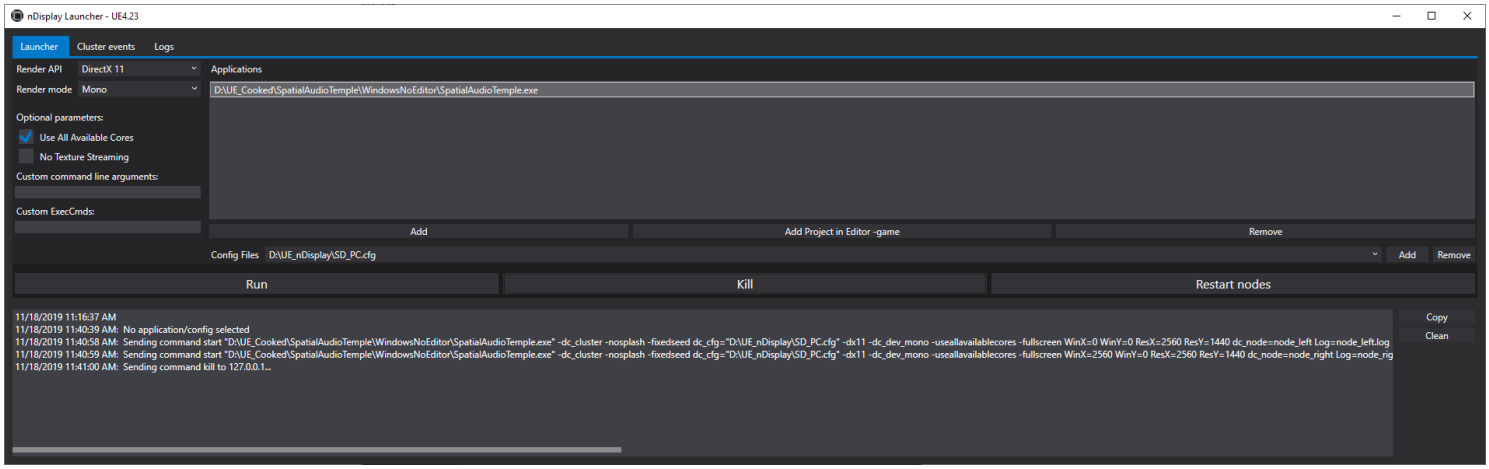

Figure 19: Screenshot of nDisplay Launcher software

Unreal Engine project-based considerations

Assuming the display cluster has already been properly set up, enabling nDisplay in a project is fairly straightforward. In many cases, you will not need to change anything about the UE4 scene and can use it as is. Other cases will require minor modifications to your setup.

There are two main use cases of nDisplay that can be described as follows:

- **Static systems** - The term *static*, in this sense, describes a setup where the nDisplay camera system does not react to outside influences like viewer interaction in relation to the display. (Note that in this case, the UE4 camera may be either still or animated.) In this common use case, where there is no specific tracking needed, you will very often be able to run the project without modification using the specified configuration file. nDisplay will automatically spawn a DisplayClusterRootActor in the UE4 scene that will follow the active UE4 camera with no offset or changes required whatsoever.
- **Head-tracked systems** - For systems with head tracking (e.g. ART, Vicon), use the VRPN server tracking system specified in the configuration file. This is how most CAVEs are implemented. In this use case, the real-time adjustments in terms of position and orientation will be provided automatically to make the display image logically correspond to the viewer's point of view.

In any case, you should avoid manually modifying the UE4 camera to make it correspond to the viewer's physical point of view in relation to the display. Typical projects have many cameras (e.g. for gameplay, animation, and cinematics), and it would be difficult to manually adjust each and every one of them correctly.

If you need to adjust the UE4 camera view, the suggested workflow is to add the DisplayClusterRootActor somewhere in your project with local offsets and rotations applied to the DisplayClusterRootComponent. Adjustments done here will only apply to the nDisplay camera system without intrusive changes to the main UE4 project camera logic.

Another quick and non-intrusive option is to simply apply those changes in terms of offset and rotation to the [camera] line within the nDisplay configuration file.

Next steps / future vision

Epic Games is committed to nDisplay technology and is excited by its endless possibilities. The volume of feedback we've received from the industry, and also from our internal team, has contributed greatly to our perspective and insight on where we see this technology evolving over time.

With the ongoing developments around nDisplay, new opportunities are now within reach such as projection mapping and display canvases of any size, shape and resolution. For the near future, we are looking to prioritize user experience and the remaining determinism aspects for the next release of nDisplay. Our long-term goal is to provide a framework where any Unreal Engine project can distribute its rendering regardless of features used in the project, including physics and gameplay logic. In other words, we aim to provide a deterministic system that works with all Unreal Engine features.

In current and upcoming releases, the plan is to make it so that any Unreal Engine project can be run and displayed with minimum changes or disruption. Although there is complexity involved under the hood, the idea should remain the same for the end user:

"Enable nDisplay, define my screen topology, and then distribute my real-time rendering."

There will always be tweaks and edge cases, but globally speaking, we want our users to focus on the creative and programming aspects of their applications while nDisplay takes care of distributing and displaying the rendering.

About this document

Author

Sevan Dalkian

Contributors

Sébastien Miglio

Sébastien Lozé

Simon Tourangeau

Vitalii Boiko

Andrey Yamashev

Editor

Michele Bousquet

Layout

Jung Kwak