EPA/600/R-22/085 | August 2022
www.epa.gov/emergency-response-research

United States
Environmental Protection
Agency

Investigation of 3D Game Engines to Support Modeling Efforts

Office of Research and Development
Homeland Security Research Program

United States Office of Research and Development
Environmental Protection Agency
Washington, D.C. 20460
August 2022
EPA/600/R-22/085

Investigation of 3D Game Engines to Support Modeling Efforts

by
Ethan Burch[1], Anne Lutes[1], Greg Osefo[1], Donna Womack[1], Timothy Boe[2], Sang Don Lee[2]

[1]RTI International
Research Triangle Park, NC 27709

[2]U.S. EPA Office of Research and Development (ORD)
Center for Environmental Solutions and Emergency Response (CESER)
Homeland Security and Materials Management Division (HSMMD)
Durham, NC 27709

Disclaimer

Any mention of trade names, manufacturers or products does not imply an endorsement by the United States Government or the U.S. Environmental Protection Agency. EPA and its employees do not endorse any commercial products, services, or enterprises.

Questions concerning this document, or its application should be addressed to:

Timothy Boe
U.S. Environmental Protection Agency
Office of Research and Development
Center for Environmental Solutions and Emergency Response
109 T.W. Alexander Dr. (MD-E-343-06)
Research Triangle Park, NC 27711
Phone 919.541.2617

Table of Contents

List of Tables

List of Figures

Abbreviations

Acronym	Definition
2D, 3D	Two- or three-dimensional
API	Application programming interface
CBRN	Chemical, biological, radiological, or nuclear
CFD	Computational fluid dynamics
CGD	Computational gas dynamics
COTS	Commercial-off-the-shelf
CPP	C++, a general-purpose programming language
CPU	Central processing unit
CUDA	Compute unified device architecture
DOTS	Data-oriented Technology Stack (Unity Engine)
ECS	Entity component system
EPA	Environmental Protection Agency
FEM	Finite element method
GPU	Graphics processing unit
IDE	Integrated development environment
SPH	Smoothed-particle hydrodynamics
VR	Virtual reality
XR	Extended Reality

Acknowledgments

Contributions of the following individuals and organizations to this report are acknowledged:

U.S. Environmental Protection Agency (EPA) Project Team

Timothy Boe
Sang Don Lee

U.S. EPA Technical Reviewers of Report

Lance Brooks
Jamie Falik

U.S. EPA Quality Assurance

Ramona Sherman

Executive Summary

Overview

Recovery following a large-scale chemical, biological, radiological, or nuclear (CBRN) incident requires a holistic approach and clear understanding of the intricate and interconnected processes associated with characterizing hazards, decontaminating affected sites, and managing resultant wastes. Without such an approach, inferior decisions may be made, which in turn could result in an undesirable outcome (e.g., increases in cost, time, and health risks).

The ability to implement full-scale disaster response exercises with minimal resources and maximum control and realism is of great interest to the emergency response community. However, in-person exercises are expensive, time consuming, difficult to organize, and limited in scope. Currently, the U.S. Environmental Protection Agency lacks modeling and decision support systems to test, train, and evaluate strategic approaches to chemical, biological, radiological, or nuclear response and cleanup scenarios outside of such large-scale demonstrations or real-world incidents. There is a significant need for developing a simulator capable of visually depicting hypothetical disaster response and recovery scenarios and using these simulations to train responders/decision makers. The Environmental Protection Agency is, therefore, evaluating the use of three-dimensional commercial-off-the-shelf (COTS) game engines for facilitating modeling, decision making, training, and exercise efforts for chemical, biological, radiological, or nuclear incidents. Today's game engines rival or exceed the capabilities of traditional research modeling platforms: they are capable of modeling, accurately and in real time, physical systems and conditions, such as object collisions and the dynamics of fluids, particles, and light. The modification of these engines to simulate a few selected scenarios/proxy events could offer significant cost savings in the development of future decision support systems and environmental modeling tools.

The purpose of this study was to synthesize existing knowledge and related research to evaluate the use of 3D COTS game engines for facilitating the modeling of four CBRN incident proxy scenarios: transport of liquids on outdoor surfaces; dispersion of particulate matter; explosive blast physics for conducting damage and impact assessments; and effects of urban geometry on radiation attenuation. The work and conclusions presented as part of this study were empirical and observational; no scientific experiments were performed.

Methodology

The study consisted of two components: (1) a literature review to identify relevant sources of information to evaluate the use of game engines for facilitating modeling efforts related to a chemical, biological, radiological, or nuclear incident; and (2) testing of possible game engine and plug-in solutions for four proxy scenarios:

- Transport of liquids on outdoor surfaces
- Dispersion of particulate matter
- Explosive blast physics for conducting damage and impact assessments
- Effects of urban geometry on radiation attenuation.

The following criteria were used to evaluate the software products capable of answering the research questions pertaining to the above proxy scenarios:

1. Publicly available commercial-off-the-shelf product.
2. Free of coding errors.

3. Supports two or more different particles interacting with other particles.
4. Simulates collisions with other geometry in the engine.
5. Quickly and reliably runs a simulation with at least two sets of 10,000 particles.
6. Integrates fluid dynamics, including physics.
7. Integrates smoke dynamics, including physics.
8. Runs in real time within a game engine and works with built-in systems.
9. Accurately simulates real-world fluid dynamics.
10. Uses particles to calculate fluid dynamics.
11. Retrieves particle information when the simulation is paused.
12. Uses different types of fluid simulation algorithms.
13. Gathers data about each particle efficiently enough to maintain the simulation and output data on a per-frame basis.
14. Uses large-scale fluid simulations.

Results

Game Engines

Game engines are the core component necessary for a game program to run properly. Core game engine components may include a rendering engine, a physics engine, sound, scripting, animation, artificial intelligence, memory management, and more. In addition to the core components of game engines, many game engine plug-ins have been developed; these are pieces of software designed to be added (plugged-in) to an existing piece of software to extend its capabilities. Because we are evaluating a use of game engines outside traditional game development, it is also of interest to what extent the different game engines have been utilized in other, non-gaming fields. Increased focus on the use of game engines for non-gaming applications should lead to increased development of plugins and modifications for those purposes.

Based on the literature search, the two most widely used open-source game engines with the greatest potential for application to chemical, biological, radiological, or nuclear incident modeling are Unity Engine and Unreal Engine. Both are widely used, free or likely to be low cost in the proposed application, offer extensive features, and have the potential to address these scenarios. They are similar in many respects but differ the most in three areas: (1) codebase (C++ vs. C#), (2) rendering customization, and (3) ease of use. However, none of these differences are likely to have a significant impact on the choice of engine. Ultimately, the choice between Unity and Unreal will depend on the availability of built-in or plug-in systems that address the specific needs of applying game engines to the simulation of chemical, biological, radiological, or nuclear events.

In addition to these two game engines, we also consider the use of a third possibility: a standalone executable software package called SPlisHSPlasH, which is not a game engine but otherwise meets all the criteria listed above.

Possible Applications

Current commercial-off-the-shelf game engine technology is not sufficiently advanced to permit real-time, accurate modeling of blast physics or radiation attenuation. However, fluid simulation technology, which can be applied to both liquid transport and particle dispersion, is progressing rapidly, and a higher degree of fidelity is currently possible for fluid simulation than for blast physics and radiation attenuation. Eight specific applications for fluid simulation were

evaluated: three for Unity, four for Unreal, and one for SPlisHSPlasH. Of these, only one, ObiFluid for Unity, can handle both liquid transport and particle dispersion, and it is the most promising of the eight applications reviewed. Therefore, we created several case studies using ObiFluid to illustrate how it might be used to model chemical, biological, radiation, and nuclear incidents.

Conclusions and Next Steps

Based on the literature review, no single commercial solution exists that addresses all four scenarios (liquid transport, particle dispersion, blast physics, and radiological attenuation). Although several potential solutions include some important aspects, ultimately critical criteria are missing from each one. Recommendations for individual scenarios are as follows:

- **Liquid Transport and Particle Dispersion** would best be modeled using ObiFluid (Unity Engine plug-in); this plug-in is the best current option for rendering particle-based fluid dynamics within a game engine.
- **Blast Physics** showing real-time destruction with accurately modeled physics is not currently possible. It would, however, be possible to model real-time destruction that is visually convincing but does not incorporate real-world physics into the physics model.
- **Radiation Attenuation** cannot currently be completely rendered in real time within a 3D game engine. Surface penetration is not possible; however, it is possible to model surface attenuation. A custom light-based system built on the Unity Engine is recommended.

Once test environments have been established based on the above recommendations, experts in chemical, biological, radiation, and nuclear incidents should evaluate each implementation's potential to address the study goals as well as the technical knowledge and skills needed to use the software effectively.

Foreword

The U.S. Environmental Protection Agency (EPA) is charged by Congress with protecting the Nation's land, air, and water resources. Under a mandate of national environmental laws, the Agency strives to formulate and implement actions leading to a compatible balance between human activities and the ability of natural systems to support and nurture life. To meet this mandate, EPA's research program is providing data and technical support for solving environmental problems today and building a science knowledge base necessary to manage our ecological resources wisely, understand how pollutants affect our health, and prevent or reduce environmental risks in the future.

The Center for Environmental Solutions and Emergency Response (CESER) within the Office of Research and Development (ORD) conducts applied, stakeholder-driven research and provides responsive technical support to help solve the Nation's environmental challenges. The Center's research focuses on innovative approaches to address environmental challenges associated with the built environment. We develop technologies and decision-support tools to help safeguard public water systems and groundwater, guide sustainable materials management, remediate sites from traditional contamination sources and emerging environmental stressors, and address potential threats from terrorism and natural disasters. CESER collaborates with both public and private sector partners to foster technologies that improve the effectiveness and reduce the cost of compliance, while anticipating emerging problems. We provide technical support to EPA regions and programs, states, tribal nations, and federal partners, and serve as the interagency liaison for EPA in homeland security research and technology. The Center is a leader in providing scientific solutions to protect human health and the environment.

The purpose of this study was to synthesize existing knowledge and research related to evaluating the use of three-dimensional commercial-off-the-shelf game engines for facilitating the modeling of four chemical, biological, radiological, or nuclear (CBRN) incident proxy scenarios. The modification of these engines to simulate CBRN scenarios could offer significant cost savings in the development of future decision support systems and environmental modeling tools when compared to in-person exercises (which are expensive, time consuming, difficult to organize, and limited in scope).

Gregory Sayles, Director

Center for Environmental Solutions and Emergency Response

1.0 Introduction

Recovery following a large-scale chemical, biological, radiological, or nuclear (CBRN) incident requires a holistic approach and clear understanding of the intricate and interconnected processes associated with characterizing hazards, decontaminating affected sites, and managing resultant wastes. Without such an approach, inferior decisions may be made, which in turn could result in an undesirable outcome (e.g., increases in cost, time, and health risks).

The ability to implement full-scale disaster response exercises with minimal resources and maximum control and realism is of great interest to the emergency response community. The significance of disaster response training and exercise activities on emergency personnel are well documented throughout literature (Alharthi et al., 2018; Hsu et al., 2013). These activities encourage teamwork, increase the value of training and equipment, and develop realistic perceptions of job risk. Emergency responder expertise is the cumulative result of periodic training and in-person exercise. The impacts of these activities are bolstered with increasing realism.

Nevertheless, in-person exercises—especially full-scale disaster exercises that walk a trainee through a CBRN event—are expensive, time consuming, difficult to organize, and limited in scope. Furthermore, the processes involved in planning and conducting exercises have remained largely the same for decades.

Currently, EPA lacks modeling and decision support systems to test, train, and evaluate strategic approaches to CBRN response and cleanup scenarios outside of large-scale demonstrations or real-world incidents. In place of in-person exercises, simulated training amplifies real-world experiences, providing a means to evaluate problem-solving and decision-making skills, technical and functional expertise, and communication and team-based competencies (Lateef, 2010). Therefore, there is a significant need for a simulator capable of visually depicting hypothetical CBRN disaster response and recovery scenarios and using these simulations to train responders/decision makers. The potential application and impact of such a simulation tool would be far-reaching: EPA could use it to evaluate decontamination methods prior to implementation in the field, develop computer-assisted strategies using artificial intelligence, and train personnel on the use of EPA modeling and decision support tools in simulated environments.

EPA is, therefore, evaluating the use of three-dimensional (3D) commercial-off-the-shelf (COTS) game engines for facilitating modeling, decision making, training, and exercise efforts for CBRN incidents. Today's 3D COTS game engines rival or exceed the capabilities of traditional research modeling platforms: they are capable of modeling—accurately and in real time—physical systems and conditions such as object collisions and the dynamics of fluids, particles, and light. The modification of these engines to simulate a few selected scenarios/proxy events to model common CBRN incidents could offer significant cost savings in the development of future decision support systems and environmental modeling tools. In addition, due to the popularity of video games, technologically advanced virtual reality (VR) hardware and software are now economically viable on a large scale.

The purpose of this study was to synthesize existing knowledge and research related to evaluating the use of 3D COTS game engines for facilitating the modeling of four CBRN incident proxy scenarios: transport of liquids on outdoor surfaces; dispersion of particulate

matter; explosive blast physics for conducting damage and impact assessments; and effects of urban geometry on radiation attenuation.

This document presents the methodology applied and the results of this study. Researchers conducted a literature review to identify available game engines and supporting packages (**Section 3**), evaluated two selected game engines (**Section 4**); assessed their modeling capabilities (**Section 5**); and identified potential emergency response applications (**Section 6**). This report describes key elements and considerations that are necessary for 3D gaming solutions, as well as additional elements that would be useful but are not critical to effective simulations. These findings provide EPA with a better understanding of considerations for future efforts that employ 3D COTS game engines for modeling environmental events.

2.0 Quality Assurance/Quality Control

The work and conclusions presented as part of this study were empirical and observational; no scientific experiments were performed. Technical area leads evaluated the quality of the information collected by this effort (i.e., secondary data) and, based on their expert opinion, determined if the information should be documented within the literature review. Collected literature was evaluated according to the simulation requirements as defined in **Section 3**. All supporting documentation of the secondary data considered worthy for inclusion are cited. However, no experimental confirmation of secondary data (e.g., accuracy, precision, representativeness, completeness, comparability) was conducted as part of this study.

Quality control was conducted concurrently with the literature review. Any literature or software and associated plug-ins that were deemed relevant to this study were then tested and evaluated by researchers. The process involved setting up test environments in two game engines—Unity Engine and Unreal Engine—for each plug-in, and within standalone software as relevant, over the course of three to six months, depending on the software. Researchers reviewed each other's work, and any literature deemed worthy could not be included until both reviewers had evaluated it. Quality for each piece of literature was evaluated and re-evaluated throughout the study duration.

3.0 Literature Review and Software Evaluation Basis

A literature review was conducted to identify relevant articles, reports, and other information to evaluate the use of 3D COTS game engines for facilitating modeling efforts related to a CBRN incident. To identify relevant literature, researchers conducted keyword searches in Google, on trade websites and message boards, and in the Unity Asset Store; reviewed sources cited in literature identified; and reviewed the project team's existing reference library.

Literature was shared among and reviewed by project team members to identify available software and plug-ins potentially relevant to modeling the four selected proxy scenarios related to CBRN incidents:

- **Liquid Transport:** Transport of liquids on outdoor surfaces
- **Particle Dispersion:** Dispersion of particulate matter

- **Blast Physics:** Explosive blast physics for conducting damage and impact assessments
- **Radiation Attenuation:** Effects of urban geometry on radiation attenuation.

The following criteria were used to evaluate the software products capable of answering the research questions pertaining to the above proxy scenarios:

1. **Publicly available for download and installation:** Game engine and add-on package and/or plug-in for modeling CBRN must be commercially available off-the-shelf for use, and not exist only within publications.
2. **Software implementations free of errors in the code when built or executed:** Required for maintaining functional software and modeling implementations.
3. **Supports two or more different particles interacting with other particles:** Required for modeling dispersion, transport, explosive blasts, and radiation attenuation.
4. **Simulates collisions with other geometry in the engine:** Allows modeled events to have an impact on other objects in the engine.
5. **Runs a simulation with at least two sets of 10,000 particles relatively quickly and reliably:** Required for maintaining simulations that can be accurately modeled and altered as fluid simulations increase in accuracy as the particle count increases.
6. **Integrates fluid dynamics, including physics:** Required for modeling liquid transport.
7. **Integrates smoke dynamics, including physics:** Required for modeling dispersion.
8. **Runs in real time within a game engine and works with built-in systems:** Required for study goals and integration with interactive software.
9. **Accurately simulates real-world fluid dynamics:** Required for study goals.
10. **Uses particles instead of meshes to calculate fluid dynamics:** Required to accurately model fluid dynamics.
11. **Retrieves particle information when the simulation is paused:** Required for outputting particle data at any given point of time.
12. **Uses different types of fluid simulation algorithms:** Allows for integration of various fluid dispersion models.
13. **Gathers data about each particle efficiently enough to maintain the simulation and output data on a per-frame basis:** Required for usability and study goals.
14. **Uses large-scale fluid simulations:** Desirable as fluid dispersion pertaining to CBRN events occurs on a large scale.

The first criterion, that the software be commercially available off-the-shelf, was absolute: developing new game engine software specifically for this use would be cost-prohibitive. Software deemed at least moderately capable of meeting the remaining study criteria was installed on computers, tested, and summarized, and relevant information is included in this report.

4.0 Game Engines

Game engines are the core component necessary for a game program to run properly. Core game engine components may include a rendering engine, a physics engine, sound, scripting, animation, artificial intelligence, memory management, and more. Game engines may offer an intuitive user interface, which may include an integrated development environment (IDE), to facilitate building video games and lets developers test revisions in rapid succession while assembling a game or simulation. Most importantly, game engines often allow developers to reuse and adapt the same engine and some of their own previously written code to produce additional games or simulations without having to create a custom engine and all new code for each new product. In addition to the core components of game engines, many game engine plug-ins have been developed; these are pieces of software designed to be added (plugged-in) to an existing piece of software to extend its capabilities.

Basic Game Engine Terminology

Game engines place basic, representative objects into an enclosed environment and then perform a variety of 3D transformations on them to create visual simulations.

The *objects* are called GameObjects in Unity and Actors in Unreal Engine.

The *environment* in which they are placed may be called a Scene (Unity), Level (Unreal), or Map (Unreal).

Within the context of game engines, it is helpful to understand, at a basic level, the difference between an "object-oriented" design and a "data-oriented" design.[1] Object-oriented design defines objects and assigns types of data and functionality to these objects, mirroring how we interact with the real world. Object-oriented design is intuitive and in widespread use; however, in the context of video games, it also makes highly inefficient use of CPU and memory, and so can impact performance and make it difficult to reuse functions. An alternative that is becoming more widely used is data-oriented design, which is focused on structuring data to align as closely as possible to how and in what sequence it will be used, doing much of the work upfront and making it possible for functions to be more general, more efficient, applicable to larger chunks of data, and reusable. This helps avoid hardware constraints and improve performance; it also takes full advantage of multicore processors now common in gaming computers.

Because we are evaluating a use of game engines outside traditional game development, it is also of interest to what extent the different game engines have facilitated their use in other, non-gaming, fields. A greater focus and commitment to unorthodox applications suggests greater availability of resources for informing such applications.

Based on the literature search, the two most widely used open-source game engines with the greatest potential for application to CBRN incident modeling are Unity Engine (https://unity.com/) and Unreal Engine (https://www.unrealengine.com/en-US/). This section compares them within the context of the goals of this study. The two are direct competitors and have many near-identical features. In addition, we consider the use of a third possibility identified in the literature search, a standalone executable software package called SPlisHSPlasH (https://www.interactive-graphics.de/SPlisHSPlasH/doc/html/index.html), which is not a game engine but otherwise meets all the criteria listed in **Section 3**.

[1] We are indebted to Jonathan Mines' piece on Medium, *Data-Oriented vs Object-Oriented Design*, March 20, 2018, for the information incorporated in this summary (https://medium.com/@jonathanmines/data-oriented-vs-object-oriented-design-50ef35a99056).

4.1 Unity Engine

Unity Engine is a game engine that can deploy to most popular operating systems and platforms. Unity's codebase is in C# with some in C++.

Unity offers traditional object-oriented design but is transitioning toward the more efficient data-oriented design (https://unity.com/roadmap/unity-platform/dots). Within Unity, this is called data-oriented technology stack (DOTS). DOTS is still in development and currently functions as an extension to the traditional Unity Engine. DOTS is intended to enable developers to take full advantage of multicore processors, transitioning development from object-oriented scripting to data-oriented scripting and avoiding hardware constraints, resulting in performance gains and better optimization.

Unity Overview

Version Used: 2020.1.17

Pricing: basic version is free; Pro is $1,800/yr/user, not clear if that would be needed. May also charge royalties.

Market Share: 48%; used for 50% of mobile games, 70% of top 1,000 mobile games

User Community: ~200K on official subreddit

Codebase: mostly C#

Open source? No

Unity is known for extending its application into fields beyond game development, such as automotive, manufacturing, robotics, architecture, film, and other fields. Unity also provides libraries and application programming interfaces (APIs) for unorthodox hardware and devices that are outside the field of game development, such as simulation of human physiology and electrocardiogram (ECG) sensors. Additionally, Unity is well documented and has a vast library of official and unofficial tutorials.

4.2 Unreal Engine

Unreal Engine is a popular open-source game engine developed by Epic Games (https://portal.productboard.com/epicgames/1-unreal-engine-public-roadmap/tabs/24-unreal-engine-4-27-summer-2021). Unreal focuses on high-fidelity and photorealistic graphics for immersive experiences, and it uses a visual programming language called Blueprints, which is arguably easier for programming novices. Unreal supports object-oriented design and its code base is entirely C++.

Unreal is primarily known for game development, although it is beginning to extend its application into the architecture and automotive industries.

Unreal Overview

Version Used: Unreal Engine 4

Pricing: free; 5% of royalties once game published; not clear how this would impact EPA's proposed use

Market Share: 13%; used for most AAA-studio game development (high-profile, non-mobile games)

User Community: ~100K on official subreddit

Codebase: C++

Open source? Yes

4.3 SPlisHSPlasH

SPlisHSPlasH (hereafter referred to as SplishSplash) is an open-source library for physics-based simulation of fluids developed by the Interactive Computer Graphics group of University of Freiburg. It is important to note that SplishSplash is not a game engine but a standalone C++-based tool that incorporates a series of different pressure solver solutions and allows for multiple types of particle representations for objects within a scene in real time; this means that different surfaces behave differently and are able to utilize different physical properties.

SplishSplash is not integrated into a game engine, nor is it offered as a plug-in for Unity or Unreal. Thus, it would have to be either used as a standalone or integrated into a game engine by creating a wrapper (a piece of software that contains, or "wraps around," another piece of software); this latter option would require an extensive level of effort. However, it offers a potential path forward, so we have included it in our review.

4.4 Comparison of Game Engines

Both Unity and Unreal are potential options for use in modeling CBRN events. They are similar in many respects, but differ the most in three areas:

- **Codebase:** Many external plug-ins and standalone tools are written in C++ instead of C#. Because Unreal's codebase and libraries are in C++, it can incorporate these natively without reducing performance. By contrast, Unity is based in C#, and while it can incorporate C++ libraries and tools, doing so requires real-time translation between the two languages. That translation takes processing power away from rendering and other tasks, thus degrading performance.
- **Rendering:** Unity's rendering engine is more customizable than Unreal's. The Unity Scriptable Rendering Pipeline allows developers to customize how the engine renders within the viewport and what rendering techniques are used. By cutting rendering features that would slow performance, this customization improves runtimes. By contrast, Unreal's rendering engine permits only limited customization, and what it does allow has less impact than Unity's more complete customization options.
- **Ease of Use:** Unity is generally considered easier to use for a beginner, having a more intuitive interface than Unreal. However, Unreal has more built-in starter resources. Unreal also does not offer the same level of documentation and extended reality (XR) support as Unity.

Both are widely used, free or likely to be low cost in the proposed application, offer extensive features, and have the potential to address these scenarios. Ultimately, the choice between Unity and Unreal will come down to the availability of built-in or plug-in systems that address the specific needs of applying game engines to the simulation of CBRN events, rather than overarching issues with the engines themselves.

5.0 Assessment of 3D Game Engines to Simulate Physical Hazards

In this section, we turn to the specifics of applying game engines to the four proxy scenarios defined in **Section 3**: liquid transport, particle dispersion, blast physics, and radiation attenuation. As both liquids and particle groups behave as fluids and are modeled similarly, liquid transport and particle dispersion are grouped together in the following section.

5.1 Simulation of Fluids (Liquid Transport and Particle Dispersion)

Liquid transport models are realized by computational physics models that have been adapted specifically for hydrodynamics. These systems are in the realm of fluid mechanics known as

computational fluid dynamics (CFD). CFD represents liquid molecule clusters as a particle, which will be affected by external physics interacting with the particle and properties of the liquid itself to determine how a liquid could theoretically interact in an environment. Particle dispersion can be modeled within the closely related field of computational gas dynamics (CGD), which is used to map gaseous matter (both particulate and vapor) and to simulate interactions within the gas, with other gases, or with other types of matter.

There are several types of CFD algorithms; one of the most useful and accessible is the smoothed particle hydrodynamics (SPH) method. SPH works within a bounding box and can simulate fluids as particles within that space, along with interactions from other solids and liquid particle types. This ability to simulate fluids as particles is important because most water in games and simulations is only the surface of the body of water, which is not sufficient to capture information on depth and collisions of particles. The SPH method can also be used to solve CGD problems by adjusting the mass of the particle and how gravity affects particles.

Various prototype packages designed to work within game engines have been partially developed to visualize models in real time for fluid simulation (both liquid transport and particle dispersion). Plug-ins for game engines have also been created that could help model both proxy scenarios. Possible reasons for developing these plug-ins include furthering the field of particle simulations, creating water simulations with accurate real time fluid dynamics, adopting a technological advantage over the competition, or making simulations available to a wider audience.

5.2 Simulation of Blast Physics

We found no documented use cases for 3D modeling of accurate blast physics in real time inside of game engines, although web-based modeling programs exist for getting values of a blast radius for both radiological- and chemical-based explosions (Wellerstein, 2020). These are typically not visualized (i.e., only exist as data models), and if they are, they are usually rendered on a 2D map or surface with rough estimates within a blast range; this would not be a high-fidelity 3D rendering. Several big-budget video games do incorporate simulated explosions, as explained below, but this is usually merely a visualization or special effect.

Visualizing explosive blast models in a 3D game engine seems possible. Most simulations for explosions are rough estimates of the blast radius, blast power, and radiation left/decay over time. With the correct calculations, the area affected by a nuclear blast can be simulated. That said, real time simulations of building destruction are not possible with current algorithms and available computational power. Destructions in game engines are generally not rendered accurately or in real time. For example, faults in walls are commonly calculated using algorithms that are not true to the actual science of how a building's structure would be affected by a blast (Stack Exchange, 2011). It would take simulations of this nature roughly 4 to 6 hours per building to render using a finite element method (van Gestel, 2011). As it stands, real time explosions and building destructions are possible within game engines so long as real-world physics are not taken into consideration.

Modeling building destruction with accurate physics is feasible outside of game engines. For example, the *Extreme Loading for Structures* software offers blast design and analysis, but this is not calculated and rendered in real time, does not allow for importing custom assets, and simulations are contained within the software and cannot be exported (Applied Science International, 2021). Based on the research for this report, modeling blast physics accurately in

real time within a game engine is not currently possible but could be rendered in a more simplified manner.

5.3 Simulation of Radiation Attenuation

The modeling technique used to calculate radiation attenuation is iterative and considers the depth of the object per pixel; this is performed in distance steps (i.e., how far away an object is from the starting point, from closest to farthest). Distance step calculations are performed sequentially and cannot move forward until the previous depth is determined. This process is problematic because at every frame, the calculation is completed per incremental distance, while the raytracing (a rendering technique for simulating light transport on objects in a 3D space) is recalculated per distance step *while* taking the previous distance step into account. Ultimately, attenuation models cannot be completed within real time simulations because of this intensive calculation process. As a result, while attenuation calculations are accurate for real-world conditions, they are not possible with real time calculations.

Both Unity and Unreal have all the core features needed to create simulations for radiological models. However, the required computational power for simulating such models is a hindrance. All model simulations require high-performance multithreading CPUs, state-of-the-art graphics processing units (GPUs), and GPU computational processing frameworks given the number of parallel computations. Additionally, the models require custom shading frameworks: both engines provide shading frameworks but not to the degree needed. As it stands, this is not possible within a game engine because this process cannot be calculated and rendered in real time. However, it may be feasible to render radiological attenuation if limited to a surface-only simulation or if other elements are scaled back.

5.4 Summary of Physical Hazard Assessment Possibilities

Current game engine technology is not sufficiently advanced to permit real time, accurate modeling of blast physics or radiation attenuation. However, fluid simulation technology, which can be applied to both liquid transport and particle dispersion, is progressing rapidly, and a higher degree of fidelity is currently possible for fluid simulation than for blast physics and radiation attenuation. It seems likely that it is only a matter of time until accurate, real time particle-based CFD is feasible within a game engine. In **Section 6**, we explore the possible applications for fluid simulation.

6.0 Possible Applications for Fluid Simulation

In this section, we discuss potential applications for CFD in Unity (**Section 6.1**) and Unreal (**Section 6.2**). In addition, we consider the use of a third possibility, a standalone executable software package called SPlisHSPlasH (**Section 6.3**), which is not a game engine but otherwise meets all the criteria listed in **Section 3**. Finally, **Section 6.4** summarizes the findings. **Table 1** summarizes the applications evaluated.

Table 1. Applications for Fluid Simulation Evaluated

Application	Section	Liquid Transport	Particle Dispersion
Unity: NVIDIA FleX	6.1.1	x	
Unity: ECS-Job System SPH	6.1.2	x	
Unity: ObiFluid	6.1.3	x	X
Unreal Water	6.2.1	x	
Unreal: CPP Fluid Particles and SPH Liquid	6.2.2	x	
Unreal: NVIDIA FleX and Cataclysm	6.2.3	x	
Unreal: NVIDIA Flow	6.2.4		X
SPlisHSPlasH	6.3	x	

6.1 Unity

Unity does not have a built-in water/fluid dynamics system, so any possible solution will be based on plug-ins. The following sections outline plug-ins that are currently available for use within Unity for modeling CFD. Subheadings indicate whether the plug-in is evaluated for liquid transport, dispersion, or both.

6.1.1 NVIDIA FleX (Liquid Transport)

FleX is a position-based fluid for real time visual effects (NVIDIA, 2018). FleX uses a unified particle representation for all object types that, according to NVIDIA, enables new effects where different simulated substances can interact with each other seamlessly. FleX is a plug-in for Unity Engine that is readily available. FleX only allows for single-threaded implementations (which creates a computational bottleneck) and is reliant on NVIDIA's scripting libraries.

FleX can render a variety of object types (including particles, fluids, gases, rigid or deformable bodies, cloth, and rope) and processes (including phase transition and adhesion). **Figure 1** shows an example of a FleX simulation involving a fluid and rigid bodies. However, for the purposes of modeling CBRN events, FleX has two significant shortcomings:

- **It cannot be used to simulate interactions between different types of fluids:** all fluid simulations are controlled by a single, unified solver, so all simulated parameters (e.g., velocity, viscosity) apply to every simulated fluid in the scene. Even though FleX supports multiple types of objects, it is ultimately limited by this inability to implement different kinds of fluid simulations at once.
- **Simulated fluids rendered with FleX do not behave realistically, even when contained in small spaces:** This is due to the relatively low limit of particles (roughly 5,000) and the requirement that all fluid simulations adhere to a unified solver, as mentioned above.

In summary, FleX cannot be used as currently developed for modeling CBRN events. If NVIDIA develops it further, increasing the particle limit and allowing more than one solver, it would then be potentially useful for modeling liquid transport, dispersion, and radiological particles with a transporting fluid, but these upgrades are not an adaptation a third party could implement.

Figure 1. NVIDIA FleX Showing Rigid Bodies Interacting within a Fluid Simulation

Source: NVIDIA (2018)

6.1.2 Unity-ECS-Job-System-SPH (Liquid Transport)

Unity-ECS-Job-System-SPH is an open-source, SPH simulation using Unity's new data-oriented design implementation, DOTS (described in **Section 4**; ECS in the plug-in name is a reference to Unity's Entity Component System, which is the core of DOTS). It exists only as a proof-of-concept for DOTS, with a single Unity Scene and a set number of particles within this single environment. The system uses simple geometry (cubes and spheres) to model movement and collisions and does not address complex simulations with multiple types of fluid interactions in real time (Montes, 2018). As a limited proof-of-concept, it does not provide a complete system for simulating liquid transport for the purposes of this project.

6.1.3 ObiFluid (Liquid Transport and Particle Dispersion)

ObiFluid is a Unity plug-in for simulating fluids and gases based on the SPH fluid simulation model for movement and physics (see **Figure 2**). It uses a rendering technique called screen space fluid, which blends particles in a space to look more like the fluid the system is trying to represent using a combination of depth information, screen space curvature flow, and noise to create the results. ObiFluid contains most, if not all, of the features needed for simulating particles being affected by liquid transport, but not radiation attenuation. This is possible because ObiFluid is based on several published physics simulations that are grounded in real-world physics, including position-based fluids, real time collision detection, and shape deformation.[2]

[2] For a comprehensive list of published academic research articles that have been used in Obi, see http://obi.virtualmethodstudio.com/references.html.

Figure 2. SPH Fluid Simulation of Water Flowing from a Faucet to a Bowl That Can Be Tilted

Source: Captured by RTI researchers within Unity Engine

In ObiFluid, users can track particle movement and retrieve collision data from particles and geometry. This differentiates ObiFluid from the other plug-ins, which tend to hide collision information behind the scenes. Collisions are essential for saving particle information when interacting with geometry and seeing the movement and velocity over time when colliding with other objects.

ObiFluid includes all the particle features needed to get an accurate representation of both water and smoke simulations. It also allows the user to put constraints on the simulation and change the fidelity of the results. Some of these constrainable values include iterations of friction, collision, density, and stretch shearing.

ObiFluid is also highly optimized for performance. Most of the plug-in uses multithreaded and GPU-based code, meaning that many of these calculations occur in parallel instead of running one calculation at a time. Calculations running concurrently allow for future scalability and performance gains, as trends in hardware development show parallel computing speed increases faster than single-core clock speeds.

The one disadvantage to this plug-in is the scale at which the simulations can run. The system is capable of processing approximately 5,000 particles before it starts to experience significant performance issues because the collisions require a significant amount of processing power. This cannot be avoided and is needed to get information from the particles. Additionally, particles are bound to the same solver implementation to collide and interact with other particles, which is a limitation similar to NVIDIA FleX and its unified solver.

A series of case studies that demonstrate the capabilities of ObiFluid are described in **Section 7**.

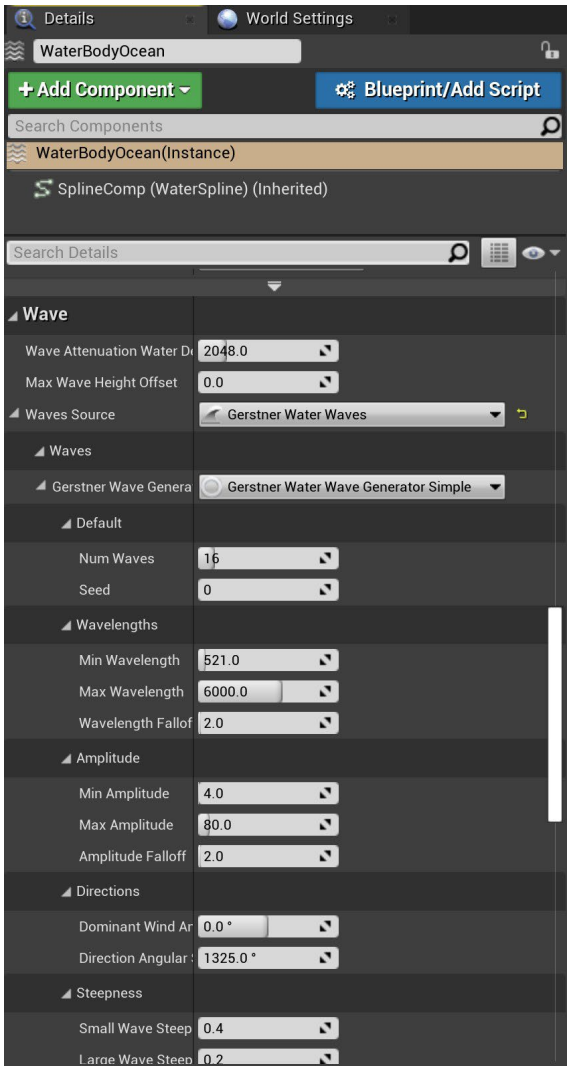
6.2 Unreal

The developers of Unreal Engine are experimenting with incorporating CFD and SPH into Unreal; they have published technical demonstrations of particle-based fluid simulations within Unreal that maintain high performance (Zhu, 2021).

6.2.1 Unreal Water (Liquid Transport)

Unreal Water (Unreal 4.26 and newer) is an integrated solution for simulating water bodies, primarily rivers, lakes, and oceans. This plug-in is a collection of modeling and rendering tools that use spline-based[3] workflows to create a unified water editing experience. It features a combined shading and rendering pipeline, as well as surface meshing that automatically supports gameplay physics and fluid simulation. Water bodies are plane- and tile-based, and waveforms are simulated on the object's mesh. What this means is that Unreal Water is easy to implement, customize, and iterate in a package that is natively integrated into the engine and maintains a high degree of performance.

Figure 3. Wave Simulation Parameters in Unreal Water



Source: Captured by RTI researchers within Unreal Engine

Because the water body objects are created using splines, users have a variety of useful options available for working with them:

- Move, rotate, scale, and duplicate
- Easily create new spline points
- Adjust water speed, depth, and audio features per point
- Access a context-sensitive menu for each point and enable visualizers to adjust water properties on the fly
- Automatically interact with and reshape the simulation landscape nondestructively.

Unreal Water comes with a built-in Gerstner wave simulation for lake and ocean waves. Each water body can have its own set of wave parameters (see **Figure 3**). Additionally, these parameters can be saved as a water wave asset and subsequently applied to multiple bodies of water. The Water plug-in can affect wave simulations by attenuating water depth, the number of waves, wave height, steepness, and several additional parameters.

In games, the fluid simulation tool can interact with characters, vehicles, and weapons. This interaction adds to the realism and feel of the game world by creating ripples, splashing, and foam effects. Forces are applied to the fluid simulation using force impulses that are controlled on a per-object basis. For example, objects can apply force to fluid simulations as they pass

[3] Splines are a type of mathematical curve. In Unreal Water, each point on the spline is represented with an interactable node for user control.

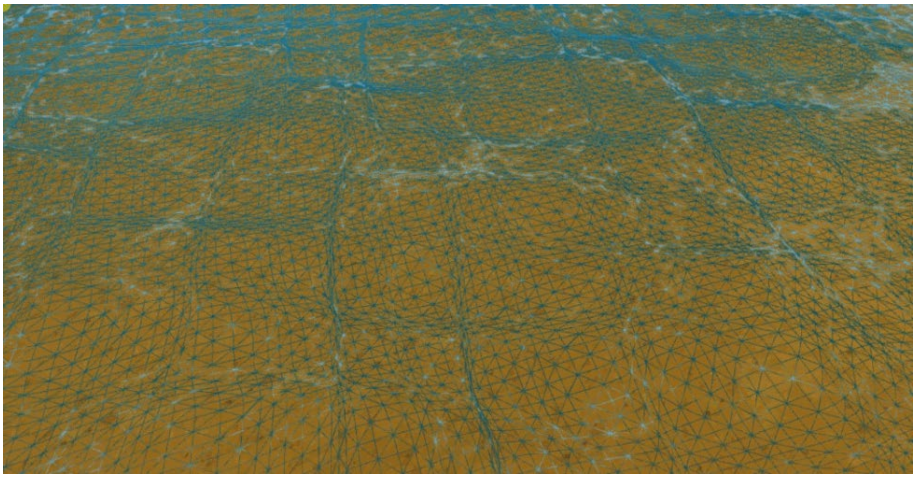
through the water, projectiles can create splashes, and ripples can reflect off the shoreline and be affected by river flow maps.

The plug-in includes two sets of equations for modeling the flow below a pressure surface in a fluid: Shallow Water and the less expansive Ripple Solver. The main difference between the two within Unreal is that Shallow Water can render sea foam and simulate water bodies draining into other water bodies downhill. Ripple Solver cannot simulate water movement to the same degree as Shallow Water, but it is substantially more stable within the engine.

All ocean, lake, and river water bodies are rendered using a single water mesh object. This special object automatically generates the needed mesh for each water body in a simulation based on their splines so that overlapping water body objects share the same mesh and water flows seamlessly across the transition.

The level of detail and all wave animations are handled using a quadtree structure (i.e., a data structure used to partition a 2D space by recursively subdividing it into four quadrants) and traversing it in each frame to generate an optimized set of visible tiles. **Figure 4** demonstrates the concept inside Unreal Engine.

Figure 4. Unreal Water with Enabled Quadtree Structure



Source: Captured by RTI researchers within Unreal Engine

Water rendering contains several properties for customizing the look and feel of water body objects:

- An underwater post-process material is applied based on the location of the user's camera to allow for partial and full submersion in a water body and simulates underwater light diffraction.
- Specified transitions between rivers and other water bodies are material driven. Transition materials can be specified on each river object to be automatically assigned to transitions between rivers and lakes and between rivers and oceans.
- Rendering custom caustic materials (the envelope of rays that are reflected or refracted by a topological space) is possible via caustics generation tools and can be applied to shallow water surfaces.

Although the water mesh object can be customized and controlled by developers, the properties for customizing Unreal Water are too expansive to discuss in this document and can be found in the Unreal Engine documentation.[4]

Ultimately, Unreal Water is not particle- or physics-based, which are requirements for accurately simulating CFD. Water is rendered on a flat plane, and all wave motions are simulated visually. Any physical interactions would require manually scripting via the Unreal visual scripting language (Blueprints) or C++ and would be additional components to the water bodies. Water properties are not true to life for many reasons, in part because Unreal measures and renders all game objects on a significantly smaller scale than reality. For example, one experiment included an ocean with a depth of 2.5 meters at the deepest point. It is important to note that this is not a flaw in the engine or the water plug-in; rather, this is a feature to ensure the system performs well in a gaming context. Additionally, it does not allow different kinds of fluids; this system is built specifically for water bodies. Unreal Water is the easiest option to set up and iterate on, but it is not viable for this project because any physical components of water movement would not be part of the water simulation.

6.2.2 CPP Fluid Particles and SPHLiquid (Liquid Transport)

CPP Fluid Particles (in which CPP stands for C++) is an open-source implementation built by one researcher based on several SPH papers (Bender and Koshier, 2015; Macklin and Müller, 2013; Becker and Teschner, 2013; Akinci et al., 2013; He et al., 2014) adapted to Unreal by a second researcher. It uses C++ and CUDA (a parallel computing platform developed by NVIDIA that enables software programs to perform calculations using both the CPU and GPU). SPHLiquid is an open-source Unreal Engine plug-in for CPP Fluid Particles and requires a custom version of CPP Fluid Particles. Put simply, this can theoretically model CFD within Unreal Engine.

SPHLiquid is ultimately not a solution for this project because it requires building the custom version of CPP Fluid Particles, which is beyond both the available level of effort for this project and the experience of the project team. In addition, it offers no instructions on how to integrate it into Unreal Engine once built. More importantly, the time required to make this a functional plug-in renders it unusable. There is also no guarantee that once built this solution would perform well, be flexible, and incorporate the required project parameters.

6.2.3 NVIDIA FleX and Cataclysm (Liquid Transport)

NVIDIA offered several plug-ins for Unreal Engine via NVIDIA GameWorks (a collection of game engine plug-ins centered around simulating water, smoke, and more) including FleX and Cataclysm. Although NVIDIA has taken GameWorks offline, it still provides documentation and instructions for accessing their GameWorks repository (https://developer.nvidia.com/gameworks-source-github).

FleX is a particle-based simulation technique for real time visual effects (**Figure 5**). Unlike the Unity version, which is a plug-in, FleX for Unreal requires installing a custom version of the engine that is no longer available from NVIDIA's repositories. We were able to install a backup of FleX and the needed custom version of Unreal via an independent repository for the purposes of this review, but that is not a practical long-term approach. FleX uses a unified particle

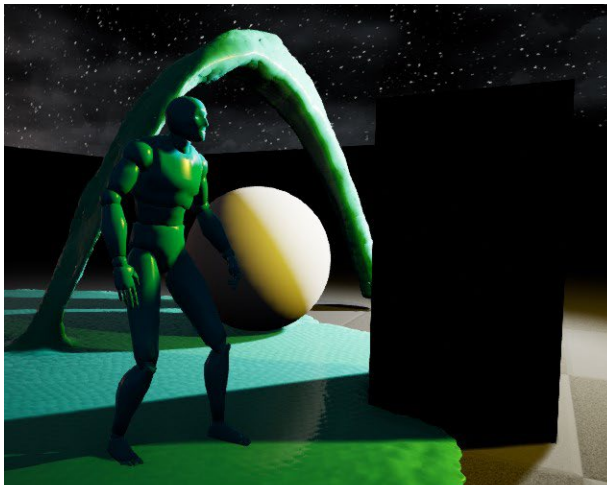
[4] https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/Water/

representation for all object types; according to NVIDIA, this feature enables new effects where different simulated substances can interact with each other seamlessly.

This version of FleX is substantially more efficient than the Unity version and offers more sample levels as well. That said, the main issues with FleX for Unreal are the same as the Unity Engine version—it cannot be used to simulate interactions between different types of fluids, because all fluid simulations are controlled by a unified solver, and the relative low limit on the number of particles means simulated fluids rendered with FleX do not behave realistically. Ultimately, FleX is an outdated rendering solution from 2013–2014. For these reasons, NVIDIA FleX is also untenable on Unreal.

Cataclysm was a technical demonstration designed to reach the scale of water simulation needed to flood a city with realistic visuals (NVIDIA, 2016). According to NVIDIA, Cataclysm can simulate up to 2 million liquid particles in real time. However, Cataclysm was never intended for full release, and the demonstration version is not available for download. Additionally, even if Cataclysm were available, it would require extremely powerful hardware. Therefore, Cataclysm is not a solution for this project.

Figure 5. NVIDIA FleX Running in Unreal Engine



Source: Captured by RTI researchers within Unreal Engine

Figure 6. NVIDIA Flow Smoke Simulation Enveloping a Sphere

Source: Captured by RTI researchers within Unreal Engine

6.2.4 NVIDIA Flow (Particle Dispersion)

Flow is NVIDIA's offering for smoke, fire, and combustible fluid simulations. Flow is available as a standalone executable simulation or as part of the same custom version of Unreal Engine with FleX. Flow uses voxels, values on a regular grid in 3D space that are commonly used to represent terrain in games and simulations. The simulation data are stored on small textures within the grid and exported to be rendered.

Flow is visually impressive: simulations behave and interact with objects realistically (**Figure 6**). It is also customizable and straightforward to set up. If Flow were available to use with Unity, then Flow, coupled with ObiFluid, would be a strong option due to Flow's robust smoke simulations and ObiFluid's CFD prowess. However, Flow is only available as a standalone executable or within a custom version of Unreal Engine that is no longer supported by NVIDIA.

Integration of Flow into Unity Engine would require a significant level of effort, and ultimately, the entire development team cannot switch to a new engine for one project and one plug-in.

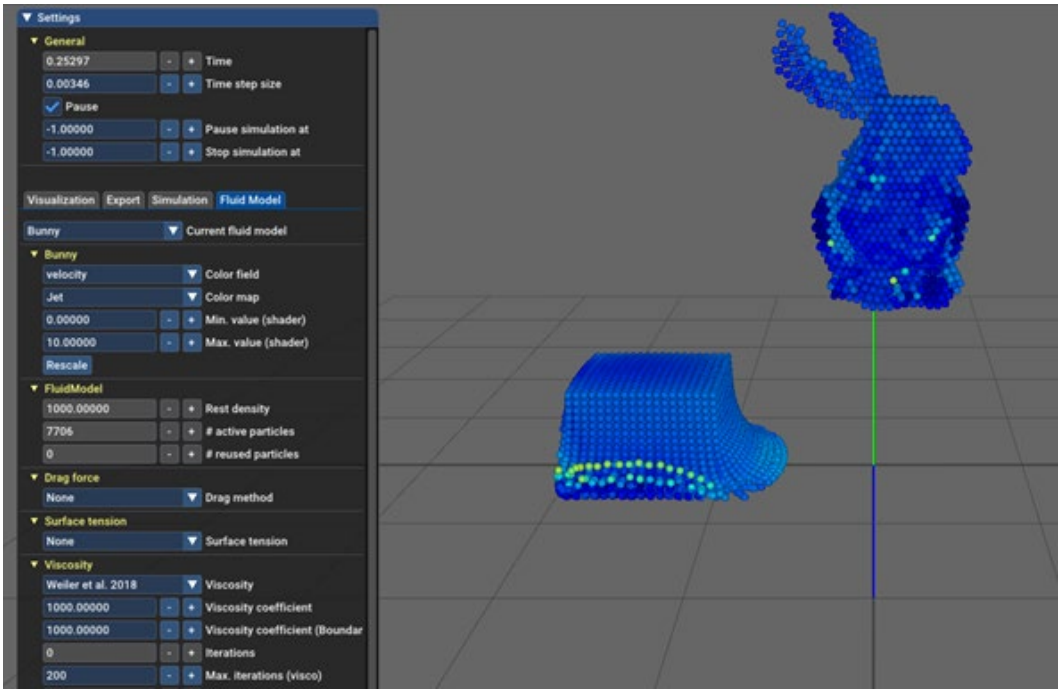
6.3 SPlisHSPlasH (Liquid Transport)

SplishSplash is based on the SPH method and incorporates state-of-the-art pressure solvers to affect fluid behaviors (Bender et al., 2020). SplishSplash may be an ideal tool for liquid transport. It uses a wide variety of particle solvers that are selectable on the fly by the user and backed by research; it models several types of fluid dynamics at runtime; fluid simulations are highly customizable, with multiple fluid properties exposed and editable in an easy-to-use user interface; and scenes are based on JSON (JavaScript Object Notation) files, a type of file that stores simple data structures and objects. The use of JSON means that writing new simulations is extremely easy and fast. In addition, it can freeze the simulation and output particle information per frame. Users can select particles with a mouse click to highlight specific particles, and information on each particle is displayed in the command line. SplishSplash also maintains a high level of performance and can output simulations as rendered videos.

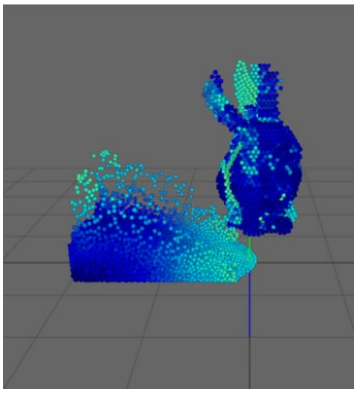
SplishSplash is exceedingly capable at CFD, with realistic particle-based simulations, which include multiple types of fluids with unique physical properties. Particles within the simulation are color-coded based on user-selectable parameters, such as velocity, to illustrate physical behaviors further. Parameters can be easily adjusted within the user interface of SplishSplash in real time, and before and after a simulation. In addition, writing new test environments was an easy process because of its use of JSON scripts. **Figure 7** shows a viscous bunny model falling into a simulated body of water that was created in under an hour.

The ideal solution for creating dynamic CFD simulations within a game engine would involve taking the time to adapt SplishSplash for either Unity or Unreal. This would be a challenging project requiring developers with experience writing C++ upwards of 1 to 2 years to complete. For that reason, it is not currently feasible but should be considered in the future.

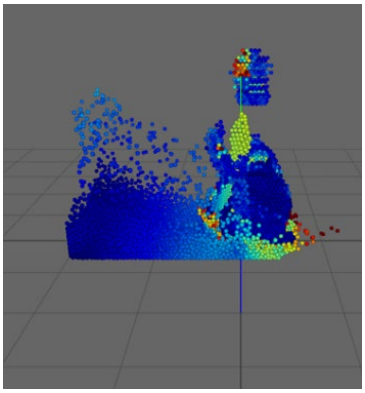
Figure 7. Images from SplishSplash Experiments

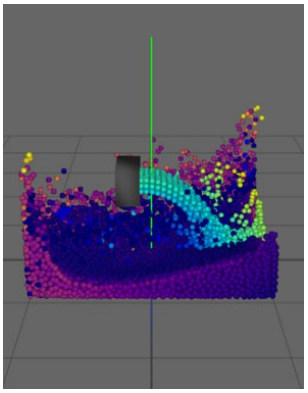


(a) The "bunny" and fluid body at simulation start. Note the parameters to the left.



(b) Bunny and fluid moments before impact.





(c) Bunny and fluid at point of impact; red, orange, and magenta particles indicate high velocity.

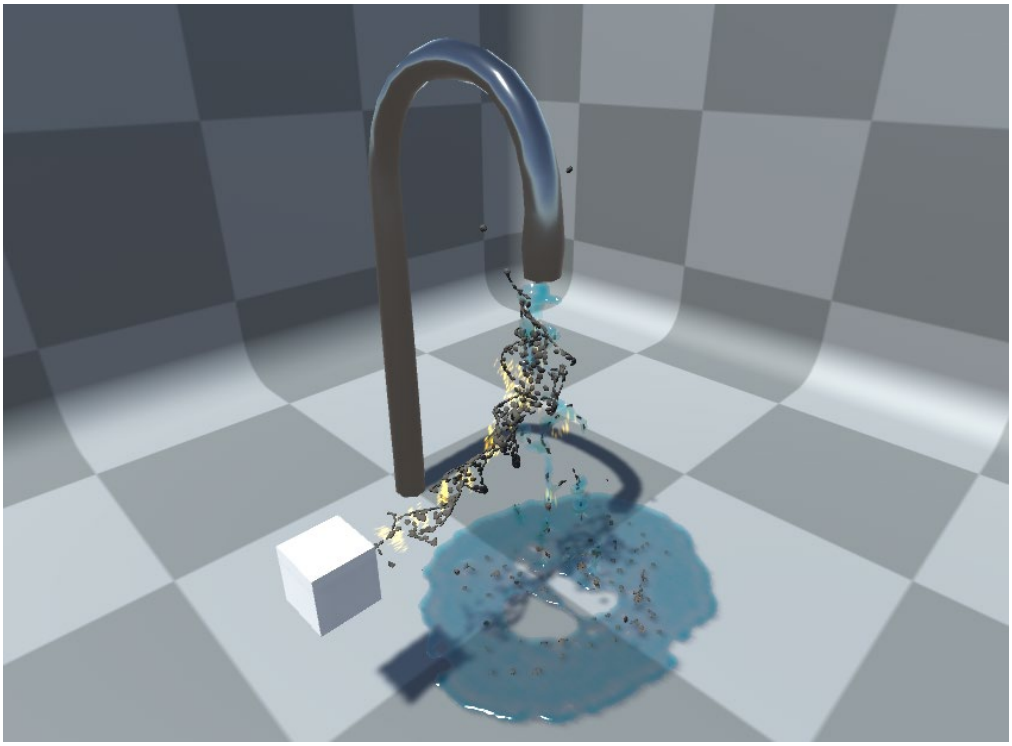
7.0 Case Studies

The most promising of the applications reviewed for simulating CBRN incidents is ObiFluid for Unity. This section presents case studies that demonstrate possible implementations of ObiFluid for the four scenarios considered here. Some are sample scenarios that come packaged with ObiFluid and were slightly modified for demonstration.[5] The case studies were developed using a high-end computer (e.g., 2.2 GHz CPU, 32 GB RAM) with a GPU.

7.1 Particle Dispersion (Smoke and Water)

This dispersion prototype (**Figure 8**) demonstrates how sediment is carried by moving water, including deposition of particles. This prototype has two particle systems at play—water falls from the faucet, pooling on the floor, and smoke particles emit from the white cube to the left. As the smoke is emitted from the cube, the particles follow a path that causes interactions with the water; some of the particles are then deposited at the base of the environment due to the simulated forces from the water falling. Remaining smoke particles continue to rise in the environment, and are either dampened by the fluid simulation, or collide with the faucet itself. This prototype demonstrates how dispersion moves sediment from one location to another; unfortunately, for now this is only possible on a smaller scale due to current limitations with ObiFluid and real time fluid simulations, but we could explore scaling other GameObjects down to a smaller size to see if that is visually convincing.

Figure 8. Dispersion of Simulated Smoke Particles from Water as it Pours from the Faucet



[5] Videos of all case study prototypes can be found at [public link to be added].

7.2 Liquid Transport

7.2.1 Fluid Maze

Fluid Maze is the most gamified example scenario in ObiFluid (**Figure 9**). Users are tasked with using the A and D keys on their keyboard to move fluid particles to the end of a maze with the highest level "purity" possible. The challenge (aside from moving the liquid, which behaves realistically) comes from orange and green cubes in the maze that tint the fluid with either an orange or green color, lowering the fluid's overall purity percentage, as shown in the upper right of the user interface. Fluid Maze can be seen as a rudimentary demonstration of liquid transport—the fluid simulation is collecting and carrying the orange and green contaminants (visually this is just a color change of the fluid) if it passes over either of the contaminated objects. The game even accounts for this by letting users try again should their purity level fall too low upon completion.

7.2.2 Fluid Viscosity

Two examples in ObiFluid demonstrate its ability to simulate viscous fluids. The first, "Raclette" (**Figure 10a**), simulates a viscous fluid falling onto a floating pink cube that has parameters to simulate heat levels, and a second, lower cube to lower the fluid's temperature. As the fluid falls, its color changes as it interacts with the cubes as a visualization of this change in temperature, and the fluid's simulated properties change—this causes the fluid to become less viscous as it slides onto the lower cube. This demo can illustrate how dispersion can leave behind fluid bodies as it passes over a surface.

The second viscosity demo (**Figure 10b**) simulates a goo-like substance being poured over a spherical object. Unlike the Raclette demo, the fluid's viscosity does not change as it interacts with the sphere—instead it clings to the sphere before pooling at the ground. Altering the viscosity parameters changes how quickly the simulated fluid pools at the bottom. Similar to the Raclette demo, this is an example of fluid remaining on a surface.

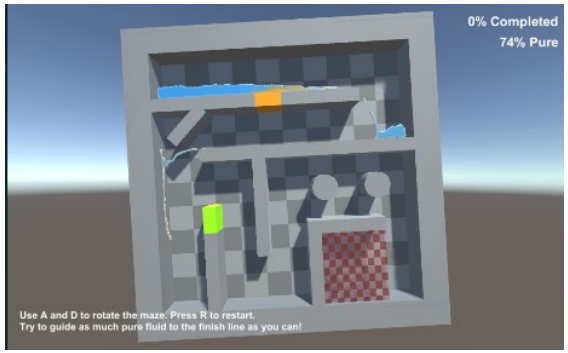
7.2.3 Fluid Mixing

The final two examples simulate fluid particles interacting and colliding as they are poured into a contained area. The first (**Figure 11a**) is similar to SplishSplash in that the fluid particles change color as collisions happen. As blue and yellow fluid bodies are poured into the environment, the demo changes the two fluids' colors based on the velocity of each particle during collisions. After a few violent moments of simulated wave crashes, the fluid bodies settle at the bottom with multiple colors.

The second fluid mixing demo (**Figure 11b**) has additional objects within the environment—two ramps, a sphere, and a cube—the latter two have simulated properties of buoyancy. The water enters in two ways—the maroon fluid passes over the ramps and the blue fluid is angled to pour off the leftmost wall. As the fluids crash into each other, the force of the collision impacts the cube, the sphere, and both fluids. The sphere and cube are realistically pushed around by the water bodies and bob up and down as determined by their buoyancy. This demo could be a good illustration for testing air filters.

Figure 9. Fluid Maze

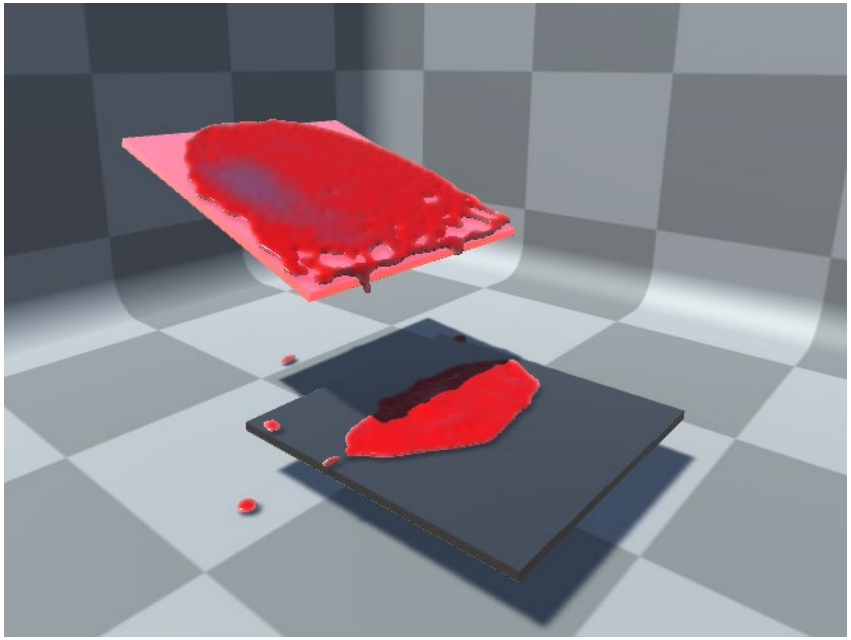
(a) Start of the scenario



(b) After moving some of the fluid over the orange "contaminant"

(c) After interacting with the green "contaminant." Note the greatly reduced purity and changes to the fluid's coloration

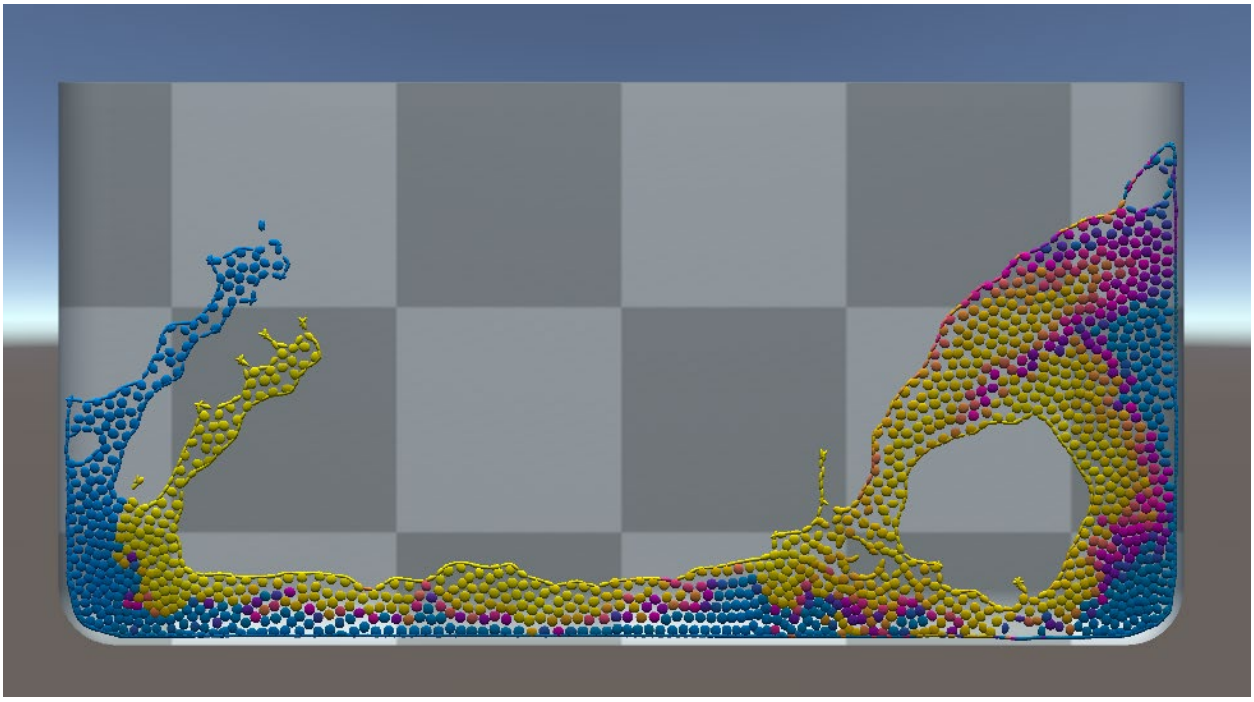
Figure 10. Fluid Viscosity



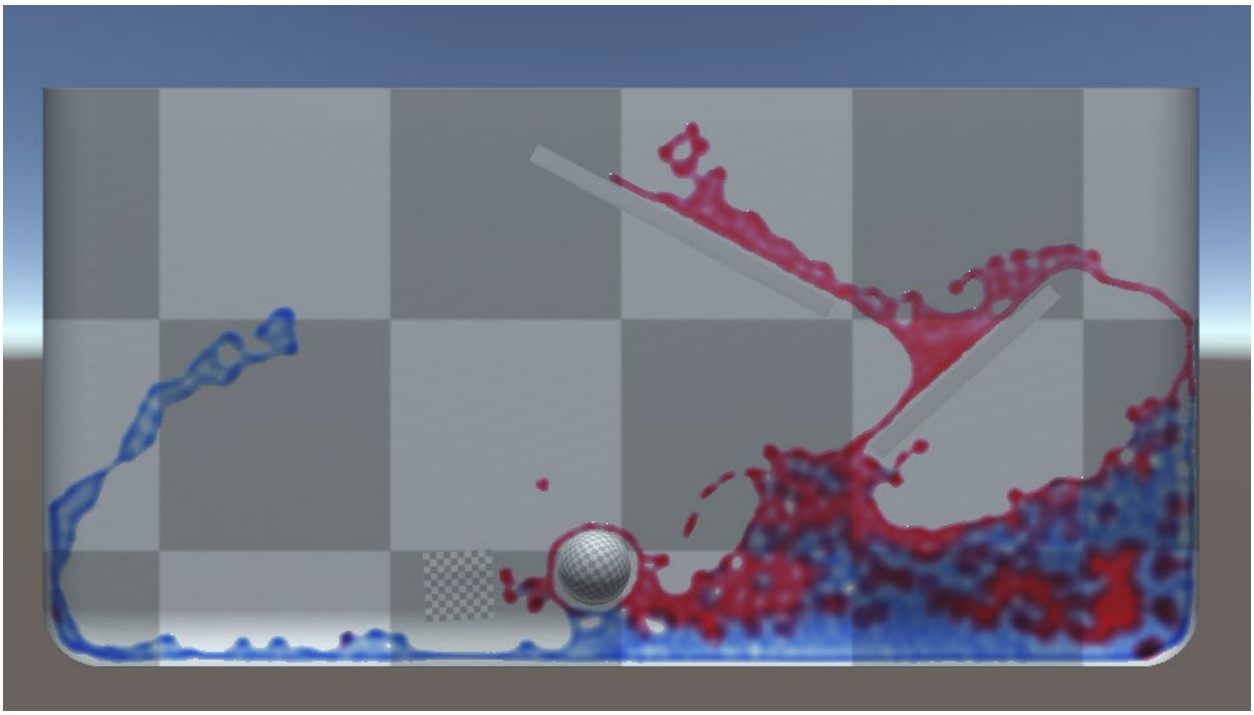
(a) Raclette demo

(b) Goo-like substance

Figure 11. Fluid Mixing



(a) Fluid particles change color as the blue and yellow fluids collide.



(b) Fluid mixing with buoyancy; cube and sphere are physically manipulated by the blue and red fluids.

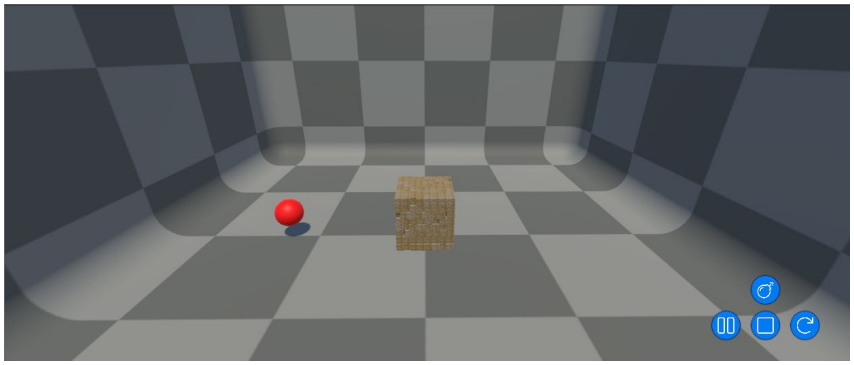
7.3 Blast Physics

Figure 12 shows a progressive series of images from the developed blast physics demo. The demo uses ObiFluid and Unity's physics capabilities to simulate a radial blast within a 3D environment. The red sphere represents the blast location and the cube in the center represents a structure that can be affected by the blast. The cube structure is composed of many soft body particle blocks (normally used for fluid simulations) and underwent a process to determine the appropriate distribution of soft body particles to represent the structure. When the bomb button is pressed (top button in the lower right of the UI, see Figure 12) a simplified blast calculation is performed from the blast location, and the cube structure is broken and scattered across the environment into separate blocks. While the demo provides a good starting point for simulating blast physics, available game technology is not currently sufficiently advanced to simulate blast physics in complex environments.

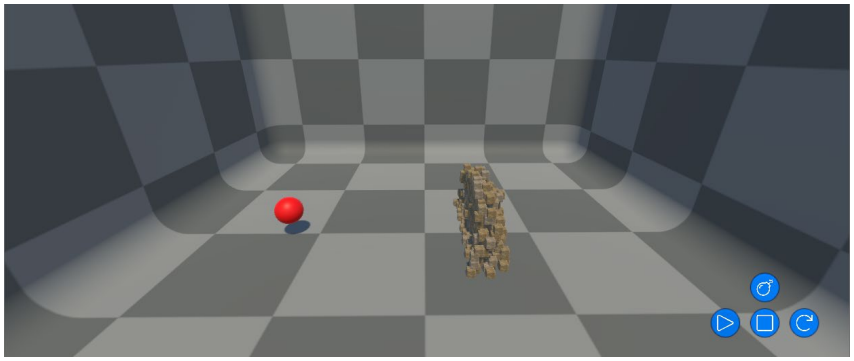
7.4 Radiation Attenuation

Figure 13 shows a before and after image from the developed radiation attenuation demo. The demo simulates radiation traveling outward from a point source in all directions. The green sphere represents the location of the radiation point source. The environment also contains various simple primitive objects that can receive radiation as well as occlude other objects from receiving radiation. The gradient displayed in the lower left of each image provides a simple colorization of the relative amount of radiation that an object receives based on its proximity to the radiation source (purple being closest to the source and light blue being furthest from the source). The inverse square law is used to attenuate radiation from the point source to each of the objects affected by radiation. This demo helps to show how current game technology can be used to effectively show radiation attenuation in a 3D environment, and future developments can build upon our work.

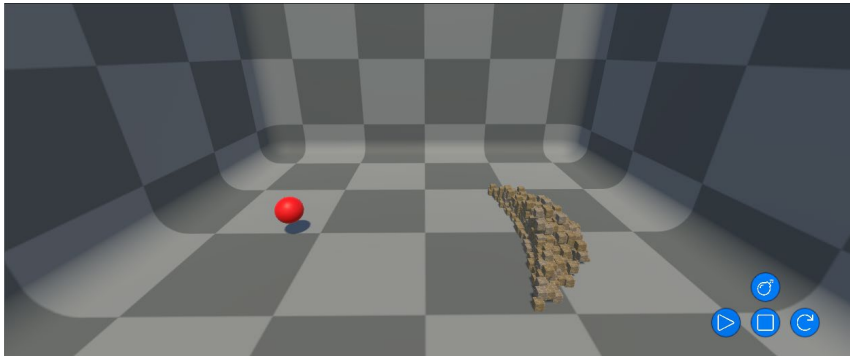
Figure 12. Blast Physics



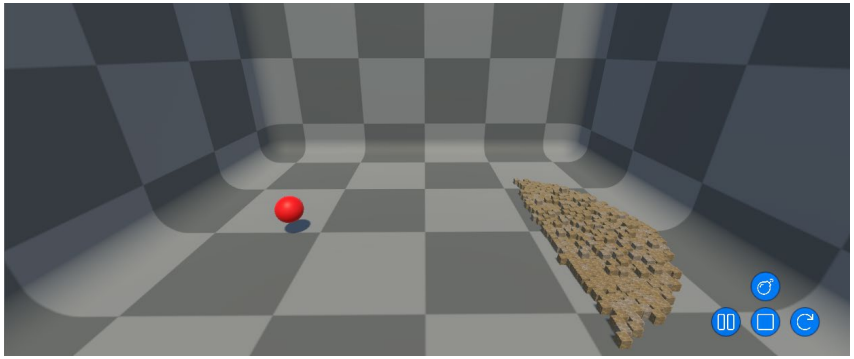
(a) The simulation environment of the blast physics demo.



(b) Snapshot of the environment during a blast.

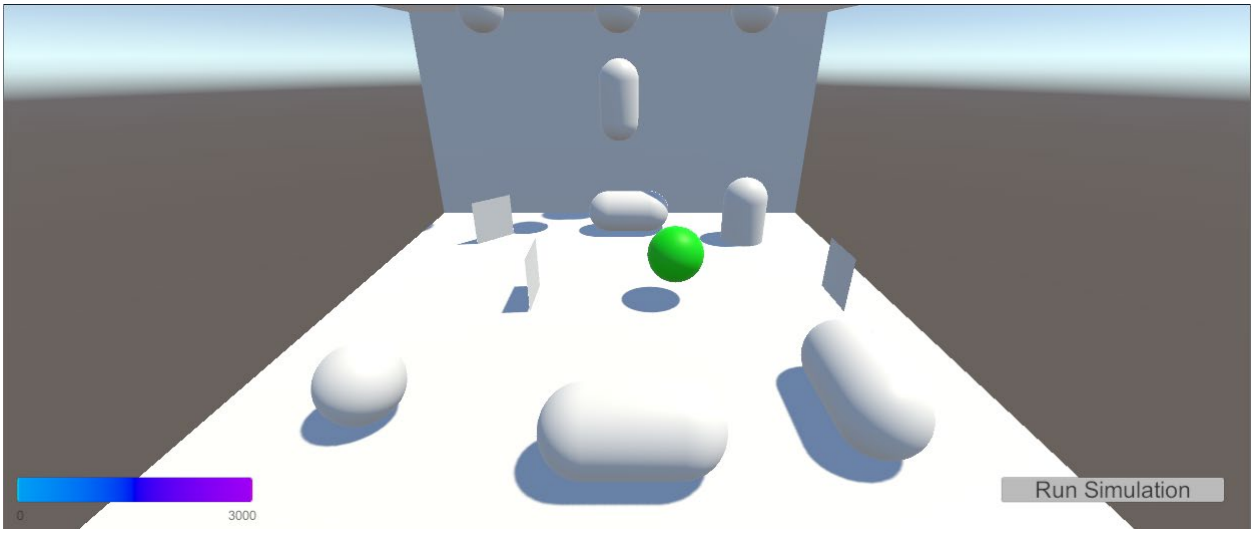


(c) Another snapshot of the environment during a blast.

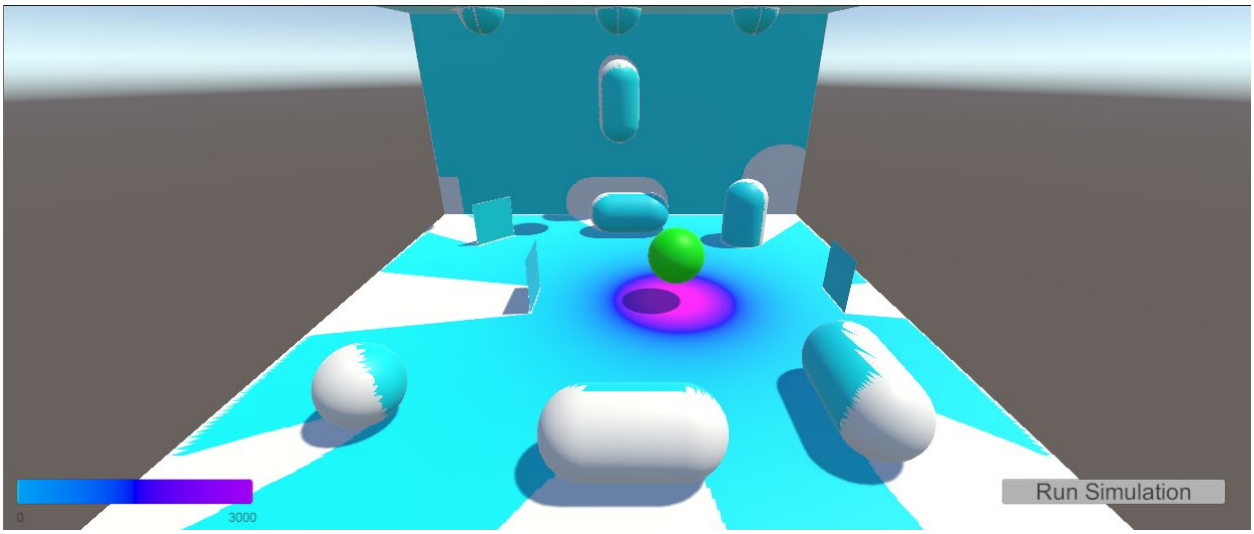


(d) The simulation environment after the blast has completed.

Figure 13. Radiation Attenuation



(a) The simulation environment of the radiation attenuation demo.



(b) Captured radiation for the environment.

8.0 Conclusions and Next Steps

Full-scale, in-person disaster training exercises are costly, time consuming, limited in scope, and have remained largely unchanged for several decades. Furthermore, EPA does not have the capabilities to adequately test, train, and evaluate strategical approaches to CBRN response and cleanup scenarios outside of large-scale demonstrations or real-world incidents. Game engines are a potential platform for modeling these simulations using built-in systems and plug-ins that can simulate collisions, fluids, particle behaviors, and lighting.

This study examined the feasibility of using two popular game engines—Unity Engine 2020.1.17 and Unreal Engine 4—to model four proxy scenarios related to CBRN incidents: (1) transport of liquids on outdoor surfaces, (2) dispersion of particulate matter, (3) explosive blast physics for conducting damage and impact assessments, and (4) effects of urban geometry on radiation attenuation.

We first evaluated general approaches to modeling the four scenarios (**Section 5**). No solutions were identified that could currently adequately address blast physics or radiation attenuation, largely due to the considerable computational demands of real-time, complex simulation and rendering for these scenarios.

For simulation of fluids (i.e., liquid transport and particle dispersion), we identified existing plug-ins that might be useful (**Section 6**). **Table 2** shows the relevant plug-ins evaluated and identifies which meet the criteria described in **Section 3**. None meet all the evaluation criteria; however, one—ObiFluid for Unity—does show significant promise.

Table 2. Engine Implementations and Plug-ins

		Unity			Unreal					
	Criterion	Unity NVIDIA FleX	Unity-ECS-Job-System-SPH	ObiFluid	Unreal Water	CPP Fluid Particles & SPH Liquid	NVIDIA Cataclysm	Unreal NVIDIA FleX	NVIDIA Flow	SplishSplash
1	Publicly available	X	X	X	X	X	X		X	X
2	Error-free	X		X	X			X	X	X
3	Two or more different particles interacting	X		X		X		X	X	X
4	Collisions with other geometry	X	X	X	X	X	X	X	X	X
5	At least two sets of 10,000 particles		X			X	X			X
6	Fluid dynamics, including physics	X	X	X	X	X	X	X		X
7	Smoke dynamics, including physics			X					X	
8	Runs in real time	X	X	X	X	X	X	X	X	
9	Accurate fluid dynamics	X	X	X	X	X	X	X	X	X
10	Particle-based fluid dynamics	X	X	X		X	X	X	X	X
11	Particle information available			X						X
12	Multiple fluid simulation algorithms									X
13	Efficient particle data management									
14	Large-scale fluid simulations						X			X

Finally, we created simple demonstrations using the most promising tools identified (**Section 7**) to highlight current capabilities and future potential for modeling each scenario. These demonstrations lead us to recommendation the following next steps:

1. **Develop a test environment using ObiFluid (Unity Engine) to build simulations of liquid transport and particle dispersion.** Liquid transport would be modeled in a simulation of two SPH-based fluid models consisting of a water body transporting radiological particles as the simulated fluids move within a contained environment.

Particle dispersion would be modeled by simulating the dispersal of particles that interact with surfaces while slowly depositing to the ground.

2. **Scope out a pared back blast physics simulation that does not incorporate real-world physics.** As noted, showing real time destruction with accurately modeled physics is not currently possible. It would, however, be possible to model real time destruction that is visually convincing but does not incorporate real-world physics into the physics model, but instead, bases destructions on random fracturing patterns without taking the material the object is comprised of into consideration. Data from existing blast impacts on buildings could then be displayed in the user interface, suggesting to users that this model incorporates real-world physics and outcomes. Eschewing visual fidelity would not necessarily lend itself to a more accurate model but exploring more pared-back approaches could provide some guidance.
3. **Scope out a custom light-based system built on the Unity Engine that would estimate the level of attenuation as a function of blast yield and distance.** Radiation attenuation cannot currently be completely rendered in real time within a 3D game engine. Surface penetration is not possible; however, it is possible to model surface attenuation.
4. **Engage experts in CBRN incidents to evaluate each implementation's potential to address the study goals.** This would include both the effectiveness of the simulation and the technical knowledge and skills needed to use the software effectively.
5. **Continue to monitor developments in game engine capabilities and improvements in computing power.** Ultimately, the raw computational power needed to accurately model CBRN incidents within a game engine is simply not within the realm of possibility at this time. However, given historical improvements in computing power, we are confident that is only a matter of time.

9.0 References

Akinci, N., G. Akinci, and M. Teschner. 2013. Versatile surface tension and adhesion for SPH fluids. *ACM Transactions on Graphics,* 32(6):182.

Alharthi, S.A., N. LaLone, A. S. Khalaf, R. Torres, L. Nacke, I. Dolgov, and Z. O. Toups. 2018. Practical insights into the design of future disaster response training simulations. In: *Proceedings of the 15th International ISCRAM Conference*, Rochester, NY.

Applied Science International. 2021. *Blast Design & Analysis*. [Software]. Available at https://www.extremeloading.com/els-applications/blast-design-analysis-software/.

Becker, M., and M. Teschner. 2007. Weakly compressible SPH for free surface flows. In *2007 ACM SIGGRAPH/Eurographics symposium on Computer Animation*.

Bender, J., and D. Koschier. 2015. Divergence-free smoothed particle hydrodynamics. In *14th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

Bender, J., T. Kugelstadt, M. Weiler, and D. Koschier. 2020. Implicit frictional boundary handling for SPH. *IEEE Transactions on Visualization and Computer Graphics* 2020:2982–93. Available at https://www.interactive-graphics.de/index.php/research/research-physically-based-animation/138-implicit-frictional-boundary-handling-for-sph.

He, X., H. Wang, F. Zhang, H. Wang, G. Wang, and K. Zhou. 2014. Robust simulation of sparsely sampled thin features in SPH-based free surface flows. *ACM Transactions on Graphics*, 34(1):7.

Hsu, E. B., Y. Li, J. D. Bayram, D. Levinson, Y. Samuel, and C. Monahan. 2013. State of virtual reality based disaster preparedness and response training. *PLOS Currents*, April 24, 2013.

Lateef, F. 2010. Simulation-based learning: Just like the real thing. *Journal of Emerging Trauma Shock*, 3(4):348–52.

Macklin, M., and M. Müller. 2013. Position based fluids. *ACM Transactions on Graphics*, 32(4):104.

Montes, L. 2018 (December 12). How to implement a fluid simulation on the CPU with Unity ECS Job System. Available at https://medium.com/@leomontes_60748/how-to-implement-a-fluid-simulation-on-the-cpu-with-unity-ecs-job-system-bf90a0f2724f.

NVIDIA. 2016 (July 25). *Cataclysm: A FLIP Solver with GPU Particles*. [Online]. Available at https://developer.nvidia.com/cataclysm-flip-solver-gpu-particles.

NVIDIA. 2018 (August 13). Video Tutorial: *FleX for Unity Plugin*. [Video]. Available at https://developer.nvidia.com/blog/video-tutorial-flex-unity-plugin/.

Stack Exchange. 2011. *How Did EA Dice Create Destructible Environments in Battlefield Bad Company 2?* [Online forum]. Available at https://gamedev.stackexchange.com/questions/15633/how-did-ea-dice-create-destructible-environments-in-battlefield-bad-company-2-an.

van Gestel, J. 2011. *Procedural Destruction of Objects for Computer Games* (Master's Thesis). Delft University of Technology, Delft, Netherlands. http://resolver.tudelft.nl/uuid:704a6d3e-1a5a-4c98-9ed5-81038ab76a7a

Wellerstein, A. 2020. *NUKEMAP*. [Online]. Available at https://nuclearsecrecy.com/nukemap/.

Zhu, A. 2021. *The Art of Illusion — Niagara Simulation Framework Overview*. [Video]. Available at http://asher.gg/?p=3014.

Definitions

Actor (Unreal Engine): an object that can be placed into a level. Actors belong to a generic class that supports 3D transformations such as translation, rotation, and scale. Actors can be created and destroyed through gameplay code (C++ or Blueprints, Unreal Engine's visual scripting language). See also GameObject (Unity Engine).

Bounding box (particle behavior): defined areas in a simulation that are checked for collisions between two objects. Simulations must remain within the bounding box.

Caustics: the envelope of rays that are reflected or refracted by a topological space. In computer graphics, this is accomplished by raytracing the possible paths of a light beam.

Computational fluid (or gas) dynamics (CFD/CGD): a group of computational physics methods to interpret how fluids (or gases) interact with themselves, other fluids (or gases), and other matter in different states.

Compute Unified Device Architecture (CUDA): a parallel computing platform developed by NVIDIA that enables software programs to perform calculations using both the CPU and GPU.

Data-oriented design: a program optimization approach used in video game development to optimize CPU cache usage and focusing on data layout and transformations.

Data-Oriented Technology Stack/DOTS (Unity Engine): in Unity Engine, a new multithreaded data-oriented technology stack (DOTS) feature that enables developers to take full advantage of multicore processors, transitioning development from object-oriented scripting to data-oriented scripting; DOTS helps avoid hardware constraints, resulting in performance gains and better optimization.

Entity Component System/ECS (Unity Engine): the core of Unity DOTS. ECS has three principal parts: (1) *entities*—the objects that populate a game, simulation, or program; (2) *components*—the data associated with entities, but organized by the data itself, rather than by entity or object; and (3) *systems*—the logic that transforms the component data from its current state to its next state (i.e., the instructions for the component data).

Finite element method (FEM): a computational method for interpreting distortion and properties of a 3D object/volume colliding and interacting with other objects/volumes. The finite element method uses real-world physics and properties of both objects and takes several hours to days to compute, depending on the length and complexity of the interactions. Thus, it cannot be used in real time.

GameObject (Unity Engine): a basic, representative object that can be placed in a level. It contains 3D transformations (translation, rotation, and scale) and can have components attached to it to give it functionality. Components are pieces of code that can be accessed by the game engine by reference to the GameObject. GameObjects can be spawned and destroyed through code (C#). See also Actor (Unreal Engine).

Gerstner wave: an exact solution for periodic surface gravity waves. It describes a progressive wave of permanent form on the surface of an incompressible fluid of infinite depth.

Level (Unreal Engine): a level is an enclosed environment that contains game objects. Also known as maps. See also scene (Unity Engine).

Level of detail (LOD): the complexity of a 3D model representation. LOD can be decreased as the model moves away from the viewer or according to other metrics such as object importance

and viewpoint-relative speed or position. LOD techniques increase the efficiency of rendering by decreasing the workload on graphics pipeline stages. The reduced visual quality of the model is often unnoticed because of the small effect on the appearance of objects when they are distant or moving fast.

Material: defines how a surface should be rendered by including references to textures, tiling, color tint, transparency, and more. The parameters available to a material depend on the shader it uses; shaders are scripts that contain the mathematical calculations and algorithms for calculating the color of each pixel rendered, based on the lighting input and the material configuration.

Mesh: a collection of polygons connected at their edges and vertices that define the 3D shape of an object.

Object-oriented design: an approach to software design that uses a programming language structure in which data and their processing methods are defined as self-contained entities called "objects". These languages provide a formal set of rules for creating and managing objects. C++ is an object-oriented programming language.

Particle behaviors:

Drag: the longitudinal retarding force exerted by air or another fluid surrounding a moving object.

Elasticity: the ability of an object or material to resume its normal shape after being stretched or compressed; stretchiness.

Friction: the force resisting the relative motion of solid surfaces, fluid layers, and material elements sliding against each other.

Surface tension: the tendency of liquid surfaces to shrink into the minimum surface area possible.

Velocity: the rate of change of an object's position with respect to a frame of reference; velocity is a function of time.

Viscosity: the measure of a fluid's resistance to deformation at a given rate. More informally, the fluid's "thickness."

Vorticity: the local spinning motion of a continuum near some point, as would be seen by an observer located at that point and traveling along with the flow. More simply, it is the twirling motion of a fluid or air.

Particle system: simulates fluids (such as liquids, smoke, and flames) by generating and animating many small 2D images in a simulated environment. In Unity Engine, this is referred to as Unity Particle System. In Unreal Engine, it is called Niagara.

Particles: many small images that are simulated and rendered by a particle system to produce a visual effect. Particles may include physical properties such as mass and velocity.

Plug-in: a piece of software that is added (plugged-in) to an existing piece of software to extend its capabilities.

Position-based fluid: a computational-based physics model for particles within fluid dynamics that is structured by the particles' positions at a given time.

Pressure solvers: a class of methods used in computational fluid dynamics for numerically solving the Navier-Stokes equations (a set of partial differential equations that describe the motions of viscous fluid substances) normally used for incompressible flows. In fluid mechanics, incompressible flows do not exhibit significant changes in fluid density, and typically have a

ratio of the speed of the flow to the speed of sound less than 0.3. By contrast, compressible flows do exhibit significant changes in fluid density, with a ratio of the speed of flow to the speed of sound greater than 0.3. Types of pressure solvers include (1) weakly compressible SPH for free surface flows; (2) predictive-corrective incompressible SPH; (3) implicit incompressible SPH; (4) divergence-free SPH; and (5) projective fluids.

Raytracing: a rendering technique for generating an image by following the path of light per pixel in the rendering viewport image and simulating light transport on objects in a 3D space.

Real time: a system in which input data are processed within milliseconds so that output is available virtually immediately as feedback. In game engines, this could include physics interactions and graphical elements that are calculated and rendered during run time and can be manipulated.

Ripple Solver: a set of equations for modeling fluid dynamics, similar to Shallow Water, albeit less expansive.

Scene (Unity Engine): an enclosed environment that contains game objects. See also Level (Unreal Engine).

Screen space fluid: a rendering technique to blend particles in a space to look more like the fluid the system is trying to represent. Uses a blend of depth information, screen space curvature flow, and noise to create the results.

Shallow Water: a set of hyperbolic partial differential equations that describe the flow below a pressure surface in a fluid. The equations are used with Coriolis forces in atmospheric and oceanic modeling as a simplification of the primitive equations of atmospheric flow.

Smoothed-particle hydrodynamics (SPH): a computational physics model within a boundary that simulates solid collisions and fluid flows with each particle.

Spline: a piecewise polynomial (parametric) curve. Splines are popular curves in computer graphics because of the simplicity of their construction, their ease and accuracy of evaluation, and their capacity to approximate complex shapes through curve fitting and interactive curve design.

Voxel: a value on a regular grid in 3D space. Common uses of voxels include representation of terrain in games and simulations.

United States
Environmental Protection
Agency

PRESORTED STANDARD
POSTAGE & FEES PAID
EPA
PERMIT NO. G-35

Office of Research and Development (8101R)
Washington, DC 20460

Official Business
Penalty for Private Use
$300