ReBots: A Drag-and-drop High-Performance Simulator for Modular and Self-Reconfigurable Robots

Thomas Collins and Wei-Min Shen

***Abstract*— A key challenge in self-reconfigurable robotics is the development and validation of complex distributed behaviors and control algorithms, particularly for large populations of modules. Physics-based, 3D simulators play a vital role in helping researchers overcome this challenge by allowing them to approximate the physical interactions of connected, autonomous robotic systems with one another and with their surrounding environments in a fast, safe, and low-cost manner that can reveal physical details that are critical to successful control. Current state-of-the-art self-reconfigurable robot simulators require users to have extensive programming (and software engineering) knowledge. Additionally, tasks such as translating specifications of real-world modules into simulated ones, creating complex configurations of modules, and designing complex environments are text-based, time-consuming, and error-prone tasks in these simulators, limiting their usefulness to quickly approximate real-world scenarios. This paper proposes ReBots, a drag-and-drop, high-performance self-reconfigurable robot simulator built on top of the Unreal Engine 4 (UE4) game engine. The mouse-and-keyboard GUI interface of ReBots allows users to rapidly prototype new modules, drag instances of them into environments, move and rotate modules, connect modules to one another, modify module properties, rotate module motors, change module behaviors, create complex and realistic environments, and run/pause/stop simulations. The results show that ReBots demonstrates high-performance and scalability of self-reconfigurable and modular robots with complex, distributed and autonomous behaviors in simulated realistic environments, including simulations of environments with up to 2000 autonomous modules physically interacting with one another.**

I. INTRODUCTION

In the field of modular, self-reconfigurable robotics, designing and validating controllers and behaviors for distributed, autonomous modules remains a challenge, particularly when considering large populations of modules. Developing such software directly on robotic hardware is error prone and can make debugging difficult, in addition to risking damage to potentially expensive prototype hardware.

Physics-based, 3D simulators address this challenge by enabling researchers to safely and, at very little cost, approximate the autonomous physical interactions of systems of modular and self-reconfigurable robots with one another and with their environments, thus revealing physical details critical to successful control.

There exist some general-purpose, physics-based, 3D simulators for modular, self-reconfigurable robots, but their lack of mouse-and-keyboard, GUI-based interfaces leads to the following limitations:

1) Using them requires extensive programming and software engineering knowledge, making them difficult for many robotics researchers, particularly those outside of a computer science background, to use effectively.
2) Simulated module specifications, environment layouts, and configurations of connected modules are specified in a text-based, manual fashion (in source code or configuration files and without visual debugging support), making the design of large-scale configurations of complex modules in realistic environments time-consuming and error-prone.

Modern game engines, such as Unreal Engine 4 (UE4) [1] – which was recently made open-source and released for free for non-commercial use – package optimized 3D rendering, state-of-the-art physics simulation middleware, and powerful environment and actor creation tools into a single toolchain designed to facilitate the creation of complex video games. UE4 exposes vast functionality through keyboard-and-mouse GUIs (making it usable to those less familiar with programming), but also provides programmers with a powerful API to deploy complex custom behavior using C++. These features make it a potentially useful platform on which to base a self-reconfigurable robot simulator. Nevertheless, using a game engine for the simulation of autonomous robot modules is not straightforward, requiring simulation designers to carefully map game engine concepts to robot simulation concepts to ensure good performance and useful simulation functionality.

This paper proposes the ReBots simulator: a high-performance, physics-based, drag-and-drop 3D simulator for modular and self-reconfigurable robots built on top of the Unreal Engine 4 (UE4) game engine. This simulator makes several unique contributions to the field:

1) ReBots provides drag-and-drop GUI visual debugging and creation tools for building new modules and modifying existing ones.
2) ReBots offers drag-and-drop GUI tools that allow users to quickly construct realistic environments and complex robot configurations (including specifying joints between modules), reducing the time and effort necessary to create useful and realistic simulations.
3) The ReBots drag-and-drop GUI can be used to run simulations of complex robotic behaviors and to change the behaviors loaded onto modules without any programming required, reducing the computer science and programming knowledge needed to use it effectively.

Using ReBots, we have demonstrated large numbers

Thomas Collins and Wei-Min Shen are with Information Sciences Institute, The University of Southern California, Los Angeles, U.S.A. collinst@usc.edu, and shen@isi.edu

[1] https://www.unrealengine.com/what-is-unreal-engine-4

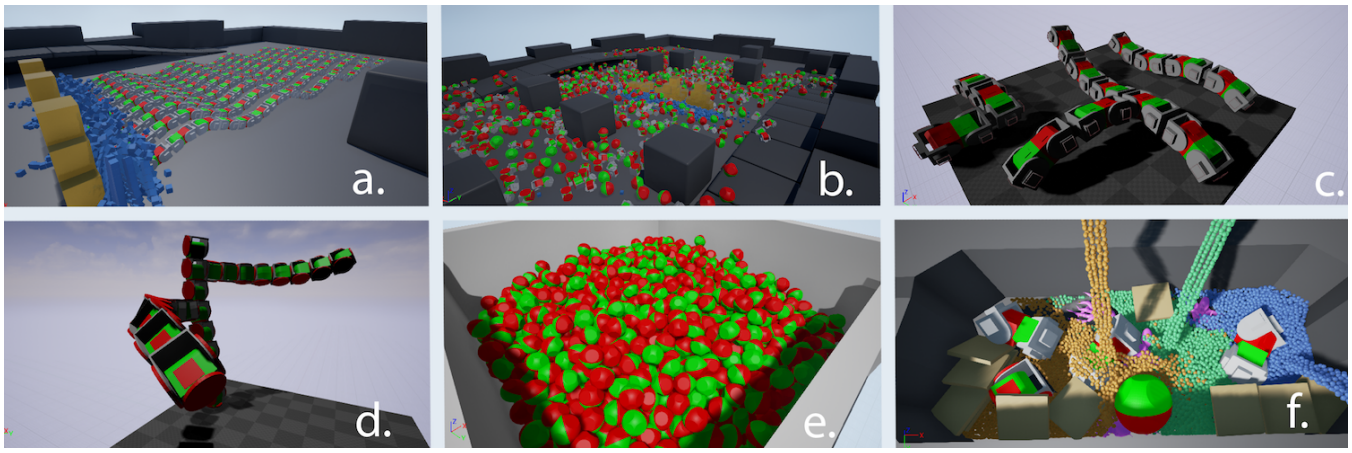

Fig. 1: Example results generated using ReBots. In a., 10, 24-SuperBot module snakes locomote into 1500 small rigid body obstacles (blue) and 18 large rigid body obstacles (gold) in a complex environment. In b., 1000 SuperBot, Smores, and RoomBot half modules randomly locomote around a complex environment. In c., three different tree structures of SuperBot modules locomote using a caterpillar gait on each branch. In d., a tree of 24 Smores modules executes a motion in its environment. In e., a pile of 2000 Roombot module halves is simulated. In f., two SuperBot modules, a Smores module, and a Roombot module half interact with 3 different types of fluid, soft bodies (purple squid), and rigid bodies (brown boxes).

(currently up to 2000) of self-reconfigurable and modular robots of different types performing complex, distributed and autonomous behaviors in realistic environments, including environments with simulated cloth, soft bodies, rope, fluid, and debris that can all dynamically and physically interact with one another and with modular robots during simulation. Some examples of autonomous robot behaviors and complex environments generated using ReBots are illustrated in Figure 1. The associated video provides many other examples.

Section II discusses related work. Section III gives some background on Unreal Engine 4. Section IV details the proposed simulator, ReBots. Section V presents results generated using the ReBots simulator. Section VI concludes with future work.

II. RELATED WORK

A number of general-purpose physics-based simulators exist for approximating the behavior of multiple physically-interacting robots of many different types in 2D and 3D environments. Many notable simulators of this type are proprietary: e.g., Webots [1], Microsoft Robotics Developer Studio[2], and V-REP[3]. However, popular open source alternatives exist: Gazebo [2], MORSE [3] (which is based on the open source Blender game engine but has very little GUI functionality), and Stage [4] (for 2.5D simulations). Newer simulators, such as SwarmSimX [5], are being developed in academic labs for future release.

Other important, but more specialized, physics-based 3D robot simulators include ARGoS [6] for swarm robots, OpenHRP[4] for humanoid robots, USARSim [7] for search and rescue simulations involving mobile robots, RoboDK[5] for industrial robots, and soccer robot simulators such as SimSpark [8] and SimRobot [9].

Though many of the simulators mentioned above have powerful GUI interfaces (including drag-and-drop functionality) that enable users to configure and run simulations quickly without extensive programming, they would require extensive modification to make them suitable for self-reconfigurable robot simulations. Crucially, they lack general mechanisms for modules to physically dock to and undock from one another and general mechanisms for modules to communicate with one another locally based on the current connections between modules. Extending these simulators to support this functionality is not at all straightforward and may not even be possible (e.g., if the source code is closed).

Finally, there have been some self-reconfigurable robot simulators proposed, including those specific to certain module types and those that are general-purpose (extensible to multiple self-reconfigurable robot types, potentially in the same simulations). Simulators specific to module types include the SuperBot [10] simulator [11], [12][6], the robot Molecule [13] simulator [14], the WeBots-based Roombot [15] simulator [16], and the MTRAN simulator [17].

To the best of our knowledge, the only two general-purpose self-reconfigurable robot simulators that have been proposed are the Unified Simulator for Self-reconfigurable Robots (USSR) [18] and our own ReMod3D [19]. The USSR simulator is well-designed, was shown to support several module types, and offers crucial, full-featured general-purpose self-reconfigurable robot simulation functionality. However, its performance is limited. It has been shown only to simulate 303 Odin [20] modules, made of simple shapes: cylinders, spheres, and cones at 1/10th real-time speed on a 2.4 GHz dual-core MacBook Pro with 4 GB

[2]http://www.microsoft.com/robotics
[3]http://www.coppeliarobotics.com
[4]https://fkanehiro.github.io/openhrp3-doc/en/about.html
[5]http://www.robodk.com

[6]Videos available at http://www.isi.edu/robots/media-superbot.html

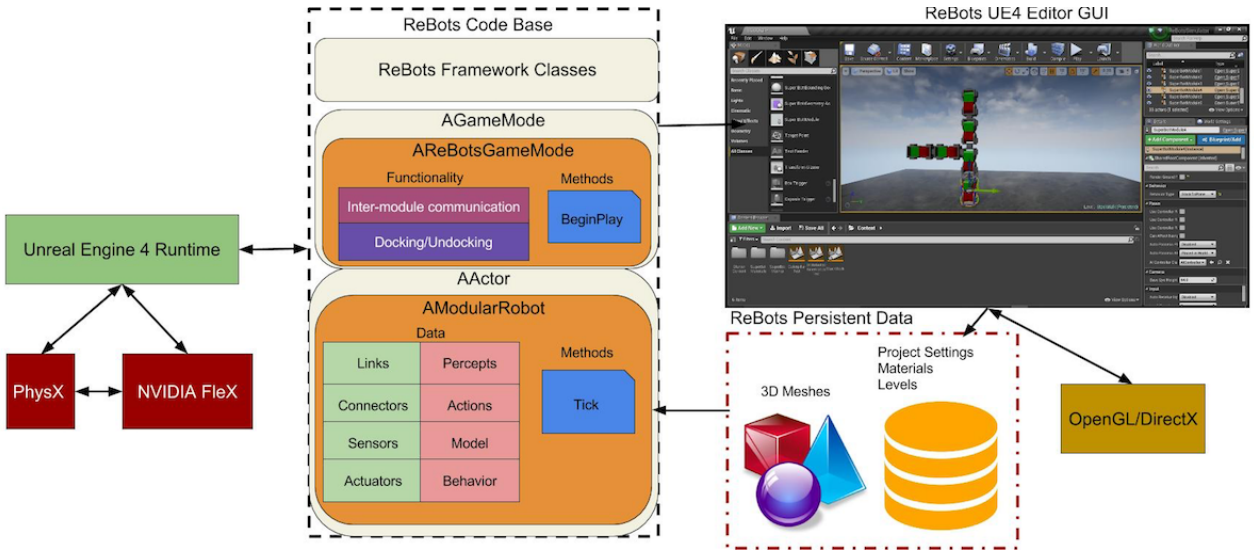

Fig. 2: The simplified architecture of the ReBots simulator, including the ReBots GUI window (ReBots UE4 Editor GUI).

of RAM. ReMod3D was shown to exceed the performance of many state-of-the-art simulators, including the USSR simulator: simulations with 1000+ connected or disconnected self-reconfigurable robots were shown to run at or over 1/10th real-time speed. Support was demonstrated for several module types, and novel environmental realism for self-reconfigurable simulations (GPU-accelerated fluid and debris) was demonstrated. Both ReMod3D and USSR lack GUI functionality to facilitate the easy creation of new modules, environments and module configurations, all of which are specified directly in source code or text-based configuration files. As a consequence, both simulators require extensive programming knowledge to be useful, and designing complex, realistic environments and module configurations remains challenging. ReBots takes key steps to overcome these difficulties, in part by using Unreal Engine 4.

III. UNREAL ENGINE 4

Unreal Engine 4 (often abbreviated UE4) is the latest major version of the professional-grade Unreal Engine game engine. The Unreal Engine has been used to create a number of popular and state-of-the-art AAA video game titles, including the Bioshock series, the Gears of War series, the Mass Effect series, Dishonored, Borderlands, and many others. At the time of this writing, Unreal Engine 4 has been released for free for academic use. The C++ source code of the engine has also been made available for customization via GitHub[7]. UE4 game development consists of extending UE4 built-in data types and functionality either using a C++ API or the Blueprints visual scripting language. ReBots was developed using the C++ API. Core ReBots functionality is integrated with UE4 using event-handling methods (e.g., $BeginPlay$, called at the beginning of simulations, and $Tick$, called at every simulation time-step) invoked by the UE4 runtime.

[7]https://github.com/EpicGames/UnrealEngine

Unreal Engine 4 makes use of the high-fidelity, multi-threaded, GPU-accelerated PhysX[8] engine for 3D physical simulations involving rigid bodies, fluids, and particle systems such as debris. The source code for PhysX has recently been released as well via GitHub[9]. UE4 can also be integrated with NVIDIA FleX [21] for unified GPU particle-based simulations of rigid bodies, soft bodies, fluids, cloth, and rope interacting in real-time. FleX simulations take place on the GPU (making them highly scalable), but FleX bodies can also interact with traditional PhysX rigid bodies. ReBots utilizes PhysX and FleX together to create high-performance simulations of robots (simulated using PhysX) in very complex environments with large numbers of obstacles and environmental elements (simulated using either PhysX or FleX). UE4 is compatible with both Mac OSX and Windows, and projects can easily be migrated between different operating systems, leading to high code reusability and making ReBots fully cross-platform. These features, along with its open source code, ease of installation, and availability for free, make it a powerful choice for the basis of a general-purpose 3D self-reconfigurable robot simulator.

IV. THE PROPOSED SIMULATOR: REBOTS

A. *ReBots Architecture*

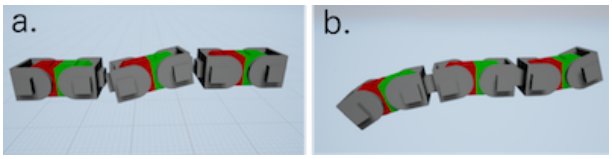

Fig. 3: Docking demonstration in ReBots. Image a. (left) shows three simulated SuperBot modules with connectors misaligned both linearly and rotationally (but within connector tolerances). Image b. (right) shows the modules being automatically projected together and docked by ReBots.

[8]https://developer.nvidia.com/gameworks-physx-overview
[9]https://github.com/NVIDIAGameWorks/PhysX-3.3

A simplified visualization of the architecture of ReBots, including the way it interfaces with the UE4 runtime and basic data types used in UE4, is given in Figure 2. ReBots is implemented as a UE4 project that consists of the ReBots Code Base (black dashed box in Figure 2) as well as persistent data storage for levels, 3D meshes (for robot bodies, obstacles, etc.), material assets for correct rendering and to encapsulate physics properties of bodies, and various project settings (red dashed box in Figure 2). The ReBots Code Base, when compiled, generates a customized version of the ReBots UE4 Editor GUI, with the custom actors, robot modules, obstacles, etc. (defined as subclasses of UE4 class $AActor$ using C++) included in the the ReBots UE4 Editor GUI for dragging and dropping into environments. During execution, the ReBots Code Base is managed by the Unreal Engine 4 Runtime, which interfaces with lower-level middleware libraries such as PhysX and NVIDIA FleX (for physics calculations) and also provides a framework for the overall game runloop, including event handling calls and runtime data type management. The ReBots Persistent Data storage is managed through the ReBots UE4 Editor GUI, which can be used to create, modify, delete etc. levels, 3D meshes, materials, and various project settings (such as rendering quality and PhysX simulation fidelity). The ReBots UE4 Editor GUI interfaces directly with the appropriate rendering API(s) (e.g., OpenGL, DirectX) for optimized rendering performance.

B. Implementation

The ReBots Code Base consists of a number of C++ framework classes that implement general-purpose self-reconfigurable simulation functionality (e.g., connection-aware message passing and module docking/undocking) and basic self-reconfigurable simulation data types (e.g., those representing modules, connectors, links, joints, messages, sensors, actuators, and behaviors) in terms of UE4 data types and inside the UE4 runloop (when necessary). The architecture of this framework is based on the *mind-and-body* architecture we presented in [19], in which robot modules have a mind (percepts, actions, world model, behavior) and a body (links, joints, sensors, actuators). This architecture was shown to support multiple modular and self-reconfigurable robot types, facilitate the development of powerful high and low-level control algorithms (behaviors), and provide a consistent and convenient interface for introducing noise into sensing and actuation. The modular and object-oriented design of this architecture makes it easy to add new modules, sensors, actuators, etc. by subclassing existing classes to customize functionality and data representation.

Unreal Engine 4 *Game Modes* (subclasses of $AReBotsGameMode$ in ReBots, which, in turn, subclass UE4 class $AGameMode$) are used to implement high-level modular, self-reconfigurable simulation functionality, e.g., inter-module communication that is aware of the connections between modules and docking/undocking of module connectors. Magnetic docking is simulated by projecting the connectors of two modules together and connecting them with a physics constraint when they become close enough to one another (see Figure 3). Geometric collision actors with no mass are attached to each connector, and docking is initiated when the geometric collision actors of two compatible connectors overlap (which generates an overlap event in the Unreal Engine 4 Runtime). This magnetic docking can be turned off or on per-connector. Modules can also initiate docking/undocking actions from their behaviors. Undocking is simulated by breaking the constraint that was placed between the modules either during docking or before the simulation began in the ReBots UE4 Editor GUI by the user. Though only magnetic docking has been implemented thus far, other game modes could easily be written to facilitate alternative forms of docking (e.g., via physical latching mechanisms simulated using PhysX rigid bodies and constraints). This is left for future work. Modules are implemented as subclasses of $AModularRobot$ (itself a subclass of $AActor$), such that they are available in the ReBots UE4 Editor GUI to be dragged and dropped into any environment. Subclasses of $AReBotsGameMode$ are also made available for selection in the ReBots UE4 Editor GUI so that any custom environment-wide functionality in them (e.g., special docking or communication logic) can be applied to any level.

C. Usage and Extension

Users interact with ReBots programmatically by extending or subclassing the provided C++ source files to create new modules, obstacles, sensors, actuators, game modes (for managing environments), etc. Module bodies are defined by importing 3D meshes (e.g., OBJ files) into the ReBots project and referencing them in the C++ code of the module. Once the sensors, actuators, actions, percepts, and world models of a robot have been defined, one or more behaviors can be developed for that module. Algorithm 1 gives an example of a "twist" behavior of SuperBot modules and illustrates the way in which module behavior code is executed within the UE4 game runloop. The $AModularRobot$ class subclasses $AActor$ which has a $Tick$ function that is called at each simulation time-step by the UE4 runtime (just before the next physics/rendering update). This function is used to read module sensors (Line 11), execute a step (tick) of the module's behavior (Line 12), and write new values to the module's actuators (Lines 13-14), among other bookkeeping operations. The behavior shown simply sets all the joint angles to 45.0 degrees (Lines 4-6), waits 100 simulation time steps (Line 7), then sets all the joint angles to -45.0 degrees. The process then repeats indefinitely, switching the joint angles every 100 time steps. The number of time steps to delay is simply set large enough to give the motors sufficient time to reach their target orientation. Custom game modes (subclasses of $AReBotsGameMode$) can be used to introduce environment-wide logic as they have a global view of the level to which they are assigned, including all the actors/modules in it. Game modes can be used modify actors, spawn new actors programmatically, parse the constraints between module docks specified by users in the ReBots

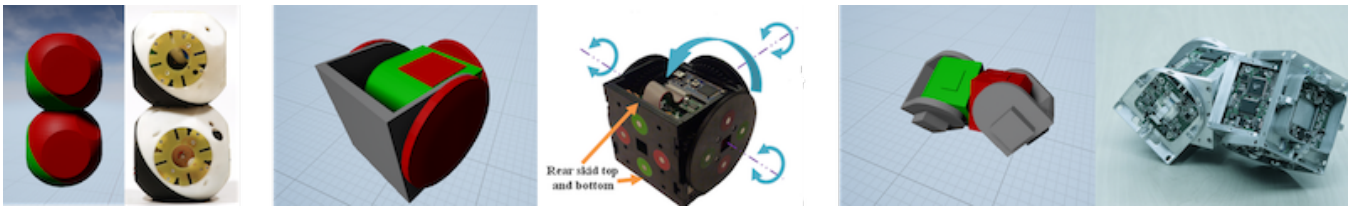

Fig. 4: The self-reconfigurable robots currently implemented in ReBots with their real-world counterparts in similar poses. Left-to-right: Roombot, Smores, SuperBot.

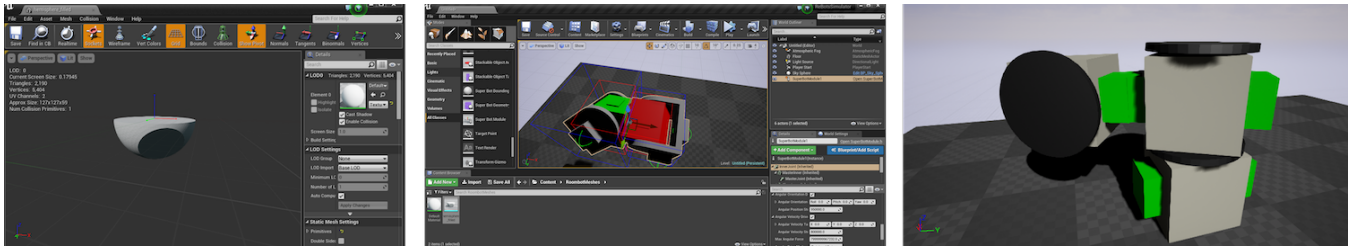

Fig. 5: Examples of GUI-based module creation and modification in ReBots. The leftmost image shows a static mesh being edited. The center image shows a SuperBot module's joints being rotated manually before simulation begins. The right image shows a configuration of three novel self-reconfigurable robots prototyped using the ReBots UE4 Editor GUI. The dark gray wheels rotate using revolute joints, while the green components slide prismatically into and away from the body.

UE4 Editor GUI (such that the modules are aware of these initial connections), override environment properties such as gravity, etc.

Algorithm 1: Example robot module behavior as executed within the overall game runloop in UE4. The code has been simplified for clarity.

```
// Class TwistBehavior
Function BehaviorTick(sVals)
    if ProgramCounter == 0 then
        TwistAngle = 45.0;
    AddActionToQueue(SET_ANGLE_0, TwistAngle);
    AddActionToQueue(SET_ANGLE_1, TwistAngle);
    AddActionToQueue(SET_ANGLE_2, TwistAngle);
    AddActionToQueue(DELAY, 100);
    TwistAngle = -TwistAngle;
    return ActionQueue;
// Class SuperBotModule
Function ModuleTick()
    sVals := ReadSensors();
    Actions := ModuleBehavior.Tick();
    for each a in Actions do
        PerformAction(a);
```

Users also interact with ReBots through the ReBots UE4 Editor GUI to build new environments, prototype new modules, populate environments with modules and other elements, and run simulations. 3D meshes imported into the simulator can be instanced in a level through dragging and dropping. By adding physics constraints between these meshes, setting joint angles/velocities, and adding external forces such as gravity, users can quickly prototype modules and simulate the module-in-progress to determine the parameters necessary to implement the module correctly in C++ source code. Such parameters – including motor gains, relative translations/rotations, center of mass adjustments, constraint projection settings – can be tricky to determine and, without a GUI, often require significant trial and error through recompilation of source code. In ReBots, this trial and error can be done visually with a mouse and keyboard without recompilation. Modules already implemented in C++ source code can also be modified similarly without recompilation, allowing users to quickly prototype changes to modules visually without needing to recompile.

Similarly, ReBots enables users to quickly create complex environments by dragging, dropping, and configuring actors, 3D meshes, modules, etc. in levels. Heightfields can be generated via a process similar to painting in Adobe Photoshop or GIMP: a brush of a certain size and shape is selected which can then be used to "paint" the heightfield. The longer the brush is held on an area, the taller that area of the heightfield becomes. By placing constraints between module connectors in the ReBots UE4 Editor GUI, users can quickly and intuitively create complex configurations of modules. Multiple actors/modules can be selected, copied, and pasted to accelerate this process (local structure within the copied actors is maintained, including constraints between them).

Simulation of configurations of modules in environments is controlled with play, pause, skip frame, and stop buttons on the ReBots UE4 Editor GUI. The behaviors compiled for each module type are available via a drop down menu when a module of that type is selected in the the ReBots UE4 Editor GUI. Modules can be independently configured with different behaviors. The simulation is automatically

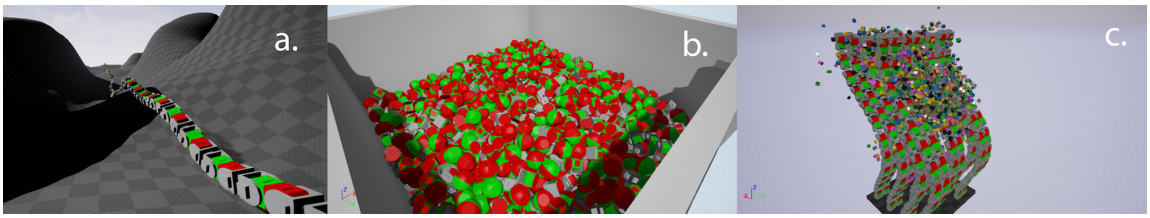

Fig. 6: Example scalability tests performed using the ReBots simulator.

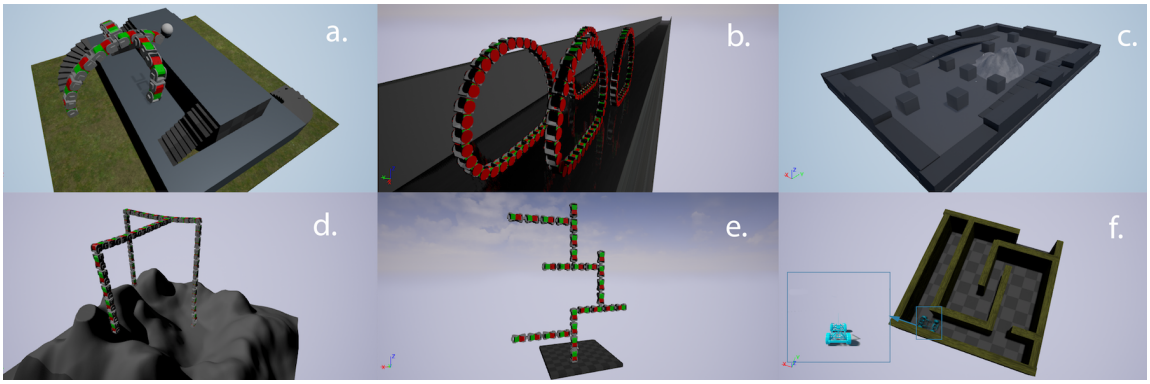

Fig. 7: Example environments and configurations of modules generated using ReBots.

reset when the stop button is pressed, allowing for easy "replaying" of the simulation multiple times (perhaps with small changes). We are currently working on developing user guides for interacting with ReBots both programmatically using C++ and through the ReBots UE4 Editor GUI.

V. RESULTS

A. Simulated Modules

Several modular and/or self-reconfigurable robots have been simulated using ReBots, including Roombot [15] modules, Smores [22] modules, and SuperBot [10] modules. Additionally, a wheeled robot with realistic, physics-based suspension and gearing has been simulated (see the associated video). The high-quality meshes for the various rigid bodies used to make up these modules were imported from OBJ files into the ReBots simulator. Each simulated SuperBot module consists of ten rigid bodies, three joints, and a total of over 2500 vertices. Roombot modules consist of four rigid bodies, three joints, and a total of over 10,000 vertices. Smores modules consist of 7 rigid bodies, four joints, and a total of over 1000 vertices. The drag-and-drop GUI of the ReBots UE4 Editor greatly simplified the implementation of these modules. Individual components of module instances (such as specific joints or specific rigid bodies) can be selected in the ReBots UE4 Editor GUI, translated, rotated, and otherwise modified without requiring recompilation. Thus, errors in module physical implementation can be quickly spotted and fixed graphically. The modified properties of the module can then be easily copied back to the C++ source code, as the properties have the same name in the C++ API as they do in the ReBots UE4 Editor GUI. Figure 4 shows the implemented simulated modules and their real-life counterparts in similar poses. Figure 5 illustrates some aspects of the module creation process in ReBots.

B. Scalability, Performance, and Environments

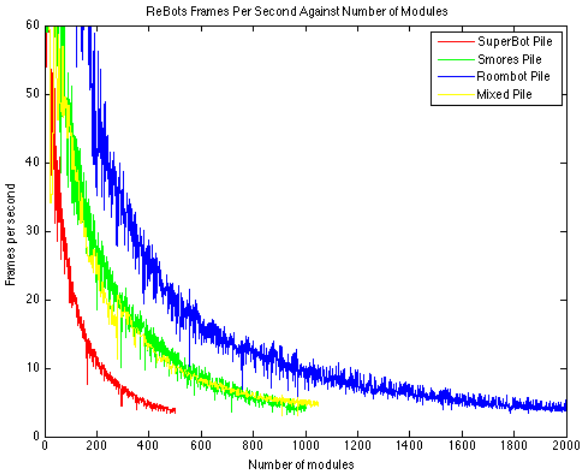

Fig. 10: Scalability of ReBots (frames per second against number of modules) when simulating piles of physically-interacting but disconnected modules in an enclosed space executing simple joint behaviors.

ReBots has demonstrated high scalability and performance, even with sophisticated configurations of modules, sophisticated environments, high-quality 3D meshes, and high-quality rendering of materials. For all scalability testing, the frame-rate was capped at 60 frames per second, and the time-step for physical simulations was 1/60 of a second. All tests were performed on a Windows PC with an Intel

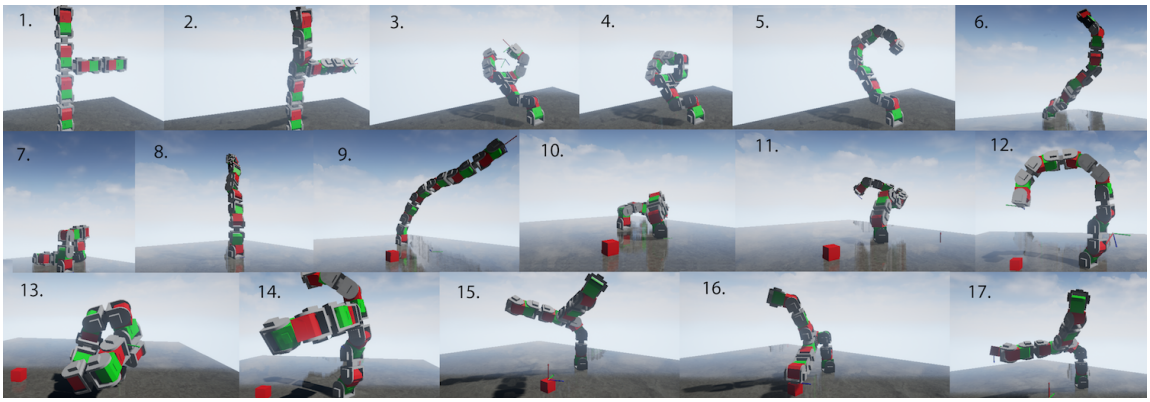

Fig. 8: A sophisticated example of self-reconfiguration and complex module behavior using SuperBot modules in ReBots.

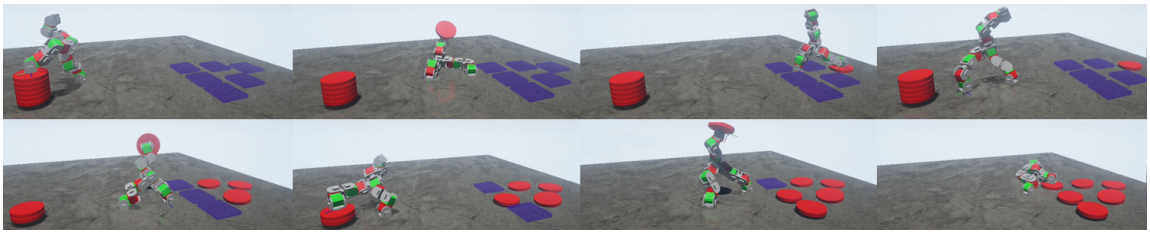

Fig. 9: A sophisticated example of locomotion, manipulation, and autonomous assembly using SuperBot modules in ReBots.

Core i7 quad-core processor, 8GB of RAM, and an NVIDIA GeForce GTX 970M GPU. All rendering settings were set to their maximum possible (highest-quality) values in UE4. Figure 6 illustrates some example scalability tests performed in ReBots. In Figure 6(a), a connected, 120-SuperBot module snake executes a caterpillar gait across a large custom heightfield environment at 19 frames per second. Simulations of 300-module snakes in the same environment executed at 8 frames per second. The largest snake tested was 500 modules in length and was simulated at 5 frames per second. In Figure 6(b), a heterogeneous pile of 1050 Smores, SuperBot, and Roombot module halves all executing simple motion gaits is simulated at 5 frames per second. In Figure 6(c), 10, 24-SuperBot module double snakes (attached front-to-back and side-to-side) sway upright into 1500 rigid body obstacles at 20 frames per second.

Figure 10 plots the frame rate of the ReBots simulator as increasing numbers of disconnected but physically-interacting modules are piled on top of one another in a relatively small, enclosed space executing simple joint behaviors. See Figure 6(b) and Figure 1(e) for the setup of two of these experiments. The red curve represents a simulation of a pile of 500 SuperBot modules executing a ”twist” behavior (Algorithm 1). The green curve represents a pile of 1000 Smores modules rotating their outer wheels at a constant velocity. The blue curve represents a pile of 2000 RoomBot module halves rotating their inner joints at a constant velocity. The yellow curve represents a mixed pile of 300 Smores modules, 250 SuperBot modules, and 500 Roombot module halves executing the behaviors described above. The curves end when no more modules can be simulated without significant freezing and stuttering. As expected, the frame rate decreases as the number of modules increases, and the more physical constraints and rigid bodies involved in a module, the fewer modules of that type can be simulated at a given frame rate. For example, SuperBot modules contained the most total rigid bodies and constraints (10 rigid bodies plus 3 physics constraints), and had, by far, the lowest scalability. On the other hand, Roombot module halves have only 2 rigid bodies and one physics constraint and were the most scalable module type. Smores modules, which have 4 physics constraints and 7 rigid bodies scale better than SuperBot modules but not as well as Roombot module halves. This scalability seems to be independent of the number of triangles and vertices in the meshes used to define the bodies themselves, likely because UE4 uses simplified collision boxes when possible to optimize performance. Note that Roombot module halves were used in order to give more variety in module complexity and still functioned as 1DOF autonomous modules.

Figure 7 gives a number of examples of complex configurations of modular and self-reconfigurable robots in complex environments generated entirely using the ReBots UE4 Editor GUI (after the modules and their behaviors had been created in C++). The joints between modules were also created graphically using the GUI. In 7(a), a manipulator of SuperBot modules (with two support arms) reaches for a sphere in a multi-level environment with stairs. In 7(b), four rolling tracks, each with 36 Smores modules, are shown in a long hallway environment. In 7(c), a complex environment with a custom heightfield in the middle of it is visualized. In 7(d), a scaffold made of SuperBot modules stands in a

custom heightfield environment. In 7(e), a complex tree of SuperBot modules is simulated. In 7(f), a wheeled robot module navigates a maze. Additional configurations and environments are shown in Figure 1.

C. Self-Reconfiguration and Autonomous Behaviors

Finally, we present results demonstrating that ReBots is both capable of facilitating the creation and simulation of complex module behaviors as well as capable of simulating general self-reconfiguration of robot modules. Figure 8 shows a tree of SuperBot modules autonomously reconfiguring itself from a tree into a long snake (1-6), stepping two times (by connecting to the ground, 7-11), reconfiguring back into a tree (12-14), and picking up a red object (15-17). Simulating this procedure required extensive use of message passing between modules (to determine kinematic structure, compute inverse kinematics, compute collision-free paths between joint configurations, etc.). Figure 9 shows a tree of 6 SuperBot modules autonomously picking up, carrying (locomoting) and placing the red objects into their appropriate goal locations (blue). These results demonstrate the simulation of high-level planning behaviors in conjunction with low-level sensors and motor control.

VI. CONCLUSIONS AND FUTURE WORK

This paper proposed a high-performance drag-and-drop simulator for autonomous modular and self-reconfigurable robots called ReBots. ReBots is built on top of the Unreal Engine 4 game engine, and takes important steps toward overcoming the high learning curve (extensive programming knowledge) and large manual coding effort required to create useful simulations in current state-of-the-art self-reconfigurable robot simulators. ReBots enables users to prototype modules, build environments, set up robot configurations, and load programs onto modules graphically with an intuitive GUI mouse-and-keyboard interface. Extensive results demonstrated the ability of ReBots to simulate complex configurations of modular and self-reconfigurable robots in complex environments (including environments with soft bodies, sand, and fluid) in a way that scaled well with the number of robots being simulated, even at maximum rendering quality settings. Future work will endeavor to automate the creation of the C++ code for robot module bodies prototyped visually in the ReBots GUI, add support for more diverse types of robot modules, sensors, and actuators, and add support for more advanced mechanical connectors.

REFERENCES

[1] O. Michel, "Cyberbotics ltd. webots tm : Professional mobile robot simulation," *Int. Journal of Advanced Robotic Systems*, vol. 1, pp. 39–42, 2004.

[2] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004, pp. 2149–2154.

[3] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular open robots simulation engine: Morse," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May, pp. 46–51.

[4] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *In Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323.

[5] J. Lächele, A. Franchi, H. H. Bülthoff, and P. Robuffo Giordano, "Swarmsimx: real-time simulation environment for multi-robot systems," in *Proceedings of the Third international conference on Simulation, Modeling, and Programming for Autonomous Robots*, ser. SIMPAR'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 375–387. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34327-8_34

[6] C. Pinciroli, V. Trianni, R. O"Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, T. Stirling, A. Gutierrez, L. Gambardella, and M. Dorigo, "Argos: A modular, multi-engine simulator for heterogeneous swarm robotics," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, Sept., pp. 5027–5034.

[7] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "Usarsim: a robot simulator for research and education," in *Robotics and Automation, 2007 IEEE International Conference on*, April, pp. 1400–1405.

[8] Y. Xu and H. Vatankhah, *SimSpark: An Open Source Robot Simulator Developed by the RoboCup Community*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 632–639. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44468-9˙59

[9] T. Laue, K. Spiess, and T. Röfer, *SimRobot – A General Physical Robot Simulator and Its Application in RoboCup*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 173–183. [Online]. Available: http://dx.doi.org/10.1007/11780519˙16

[10] B. Salemi, M. Moll, and W.-M. Shen, "SUPERBOT: A deployable, multi-functional, and modular self-reconfigurable robotic system," Beijing, China, Oct. 2006.

[11] N. Ranasinghe, J. Everist, and W.-M. Shen, "Modular robot climbers," San Diego, CA, Nov. 2007, iROS 2007 Workshop on Self-Reconfigurable Robots, Systems and Applications.

[12] H. Chiu, M. Rubenstein, and W.-M. Shen, "'deformable wheel'-a self-recovering modular rolling track," Tsukuba, Japan, Nov. 2008.

[13] K. Kotay, D. Rus, M. Vona, and C. McGray, "The self-reconfiguring robotic molecule," in *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, vol. 1, May, pp. 424–431 vol.1.

[14] K. Kotay, D. Rus, and M. Vona, "Using modular self-reconfiguring robots for locomotion," in *Experimental Robotics VII*. Springer, 2001, pp. 259–269.

[15] A. Sproewitz, A. Billard, P. Dillenbourg, and A. J. Ijspeert, "Roombots-mechanical design of self-reconfiguring modular robots for adaptive furniture," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE, 2009, pp. 4259–4264.

[16] A. Sproewitz, P. Laprade, S. Bonardi, M. Mayer, R. Moeckel, P.-A. Mudry, and A. J. Ijspeert, "Roombotstowards decentralized reconfiguration with self-reconfiguring modular robotic metamodules," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. IEEE, 2010, pp. 1126–1132.

[17] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji, "M-tran: self-reconfigurable modular robotic system," *Mechatronics, IEEE/ASME Transactions on*, vol. 7, no. 4, pp. 431–441, Dec.

[18] D. Christensen, D. Brandt, K. Stoy, and U. Schultz, "A unified simulator for self-reconfigurable robots," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, Sept., pp. 870–876.

[19] T. Collins, N. O. Ranasinghe, and W. M. Shen, "Remod3d: A high-performance simulator for autonomous, self-reconfigurable robots," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nov 2013, pp. 4281–4287.

[20] R. F. M. Garcia, K. Stoy, D. J. Christensen, and A. Lyder, "A self-reconfigurable communication network for modular robots," in *Proceedings of the 1st international conference on Robot communication and coordination*, ser. RoboComm '07. Piscataway, NJ, USA: IEEE Press, 2007, pp. 23:1–23:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1377868.1377897

[21] M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim, "Unified particle physics for real-time applications," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 104, 2014.

[22] J. Davey, N. Kwok, and M. Yim, "Emulating self-reconfigurable robots-design of the smores system," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 4464–4469.