Using the blender game engine for real-time emulation of production devices

Article *in* International Journal of Production Research · August 2011

DOI: 10.1080/00207543.2011.601772

Morten Lind
SINTEF
23 PUBLICATIONS **126** CITATIONS

International Journal of Production Research
Vol. , No. , 2011

RESEARCH ARTICLE

Using the Blender Game Engine for Real-Time Emulation of Production Devices

Morten Lind[a*] and Amund Skavhaug[b]

[a]*Norwegian University of Science and Technology,*
Dept. of Production and Quality Engineering,
7491 Trondheim, Norway

[b]*Norwegian University of Science and Technology,*
Dept. of Engineering Cybernetics,
7491 Trondheim, Norway

(*Received 00 Month 200x; final version received 00 Month 200x*)

This paper describes principles, architecture and design details for using the Blender Game Engine in real-time production device emulation. An emulation system for a real material transport and handling production installation was implemented based on this. A prototype of a distributed production control system was run on top of the device emulation system to evaluate feasibility.

A cardinal architectural principle is the clear distinction between production controllers and production devices. This principle, applicable through a flexible communication middleware, enables the implementation of portable production controllers, which execute transparently on either the emulated or the real production system.

Production system engineers may take advantage of this approach, to develop and gain confidence in complex production control solutions.

Keywords: Simulation; Distributed manufacturing control; Shop floor control; Robot applications; AGV.

1. Introduction

Distributed, autonomous production control systems, such as of a holonic or agent-based nature, have the potential to provide much higher flexibility, intelligence, and error recovery than a centralized system (Valckenaers and Van Brussel 2005). However, this comes at the price of a higher development threshold and difficulty of debugging in the run-in phase. Hall *et al.* (2005) discusses more specific obstacles to the use of agent technology

*Corresponding author. Email: morten.lind@ntnu.no

ISSN: 1366–588X print/ISSN 0020-7543 online

DOI: 10.1080/00207543.2011.601772
http://www.informaworld.com

in production control system.

Whether developing a control system for a new production installation or for an existing one, there will often be limited availability for experimenting on the real system hardware. In case of a new system with expensive hardware, the more central and expensive components will probably not be ordered until a control system is designed and validated. In case of the development of a new control system for existing production installation, the installation will probably not be recommissioned, until the new control system has been accepted.

Providing true emulators of the production hardware entities (machines, native controllers, devices, sensors, etc.) such that they can co-exist and operate in a virtual real-time world, is a large step towards developing a production control system for unavailable hardware; especially so for an agent-based control system.

1.1. *Simulation Overview*

When faced with the need of simulation for verifying or develop production control systems, it is traditional to consider *Discrete Event Simulation* (DES). A plethora of software applications and platforms exist for the purpose of simulation for manufacturing; e.g. DELMIA QUEST, FlexSim, and PySim. Smith (2003) presents an extensive classification and survey of DES in the field of manufacturing. However, DES is by its nature more suitable for simulating production control systems characterized by determinism. This is a severe restriction on control system design; especially so, when hoping to honour the request for flexibility by intelligence at the production control level.

Moon *et al.* (2006) presents a example of a combined use of a DES system, QUEST, and a robot simulation studio, DELMIA IGRIP. The body shop in an automotive factory is modelled, simulated, and analyzed. Each workstation is modelled, simulated, and optimized in detail using IGRIP, producing offline generated robot motion specifications. The resulting visual model for each workstation is then transferred to QUEST, together with resulting operation and process timings. A grand model of the whole body shop is set up in QUEST, with underlying data for all detailed models. The logic for flow of parts, buffer control, and overall logistics can then be implemented, simulated, analyzed, and optimized. This is a quite traditional separation of process control from production control. It is a time-efficient method for developing factory layout, process logic, and production logic for large batch size production. However, the method loses efficiency for simulating a production system characterized by low or single piece batch size; such as mixed or chaotic production. non-deterministic processes, or autonomous control entities.

DELMIA V5, like QUEST, is aimed at simulation, validation, and optimization of the control logic in automated production (Caie 2008). While QUEST may model the production setup and simulate an implementation of the logic of the production control system, V5 takes further steps towards realistic interaction of the modelled control system, providing realistic virtual integration of the programming aspects of PLCs, sensors, and device controllers. This amounts to a validation at a deeper level of a production control design, based on a library of many frequently used types and brands of production automation devices, such as PLCs, sensors, and device controllers. V5 aims at a maximum possible level of realistic validation in production control systems based on traditional control paradigms rooted in deterministic logic and with standardized control equipment. The work presented in this paper aims at goals of similar nature, but for paradigms of production control based on distributed autonomy and intelligence at all levels in the control system. While V5 may get close to formally validating the logic operation of deterministic production control systems, realistic real-time emulation and simulation of a control system allowing massive distribution and autonomy may yield only a qualitative and verification.

A distinctive difference which separates agent-based control systems from more traditional ones is that they are characterized by emergent rather than deterministic behaviour (Vrba and Marík 2005a). It is impossible to pose a realistic, deterministic model for a production entity controlled by a truly autonomous agent. Such a deterministic model would be a necessity for the scheduling of events in a standard DES. Real-time, realistic emulation at the device level is thus necessary, since there is no essential logic to model for a distributed, autonomous control system; there is only the actual control code to execute on the real-time emulation.

In distributed and autonomous production control, it is quite easy to argue for the advantages of time-driven, realistic simulation. Vrba and Marík (2005b) uses the term *agent-based simulation* for this type of simulation. An agent-based simulation system was described by Marík *et al.* (2005), and is a software-platform or -framework for developing and modelling an agent-based control system, and deploying it in the system of emulated production devices.

1.2. *Related Work*

Kim *et al.* (2000) implemented virtual machines for simulation of shop-floor control and operator training. They emphasize on the need for real machine emulation, providing an operator or the shop-floor control system with the realistic or real interface. However, the implementation of the virtual machines are based on the Java Web Server, Java Servlets, and communicate over HTTP/XML, which does not add up to a fast response, real-time system. Though functionally giving the appearance and true emulation of a machine, the whole range of machines requiring even soft real-time control will be excluded by such platform and communication technologies.

Park *et al.* (2009) presents an overview of principles for verification of PLC programs and commercial software for simulating production control. Though not strictly limited to PLCs, their work is focused on simulation-based verification methodology for PLC programs. The suggested method describes how to build state machines for emulation of the input-output of all production devices connected to the (real or simulated) PLC. By analyzing the logged state changes and IO-signalling, it can be established whether the PLC program satisfied the specified behaviour or if it must be improved. This resembles the methods of the present paper, but rather than building a pocket of IO-emulation for an isolated PLC, the present work tries to build a physically realistic model of all devices to be controlled by the entire (distributed) production control system. By suitably providing interfaces in software for the simulated PLC, or hardware for the real PLC, it may in fact be hosted on an emulation system based on the presented work, as part of the production control system. A further distinction between the presented work and the work of Park *et al.* (2009) is the emphasis on qualitative and quantitative verification, respectively. When using the real hardware PLC with the real production or process control program, executing with the emulated device IO state machines, this is a very thorough test of the isolated operation of the PLC program. In contrast, the approach presented in this paper may suffer from computational exhaustion in the device emulator. Additionally, the deployment systems and topology of the production control in the present work may not be the same in real-time simulation as the systems and topologies used in the production system.

Pannequin and Thomas (2010) describe *Emulica*, an architecture and implementation of a system for emulation of the production operating system on which the production control system can execute. Their framework focus on the process and flow control where parts are of transformed into products; like assembling and disassembling processes. Their emulation is at the abstract process level, providing an interface with methods of the nature of "Do the assembly" or "Move the part". Their framework is not concerned with controlling the devices and machines, performing the requests, and the related real-time aspects. In contrast, the work presented here achieves emulation of the real-time, geometric, mechanical, and control aspects of production devices.

1.3. *Outline*

The remainder of this paper is organized as follows: Section 2 gives an overview of the basic software tools and technologies used. Section 3 gives an operational overview of the demonstration system for which the emulation system was built. In Section 4 the architectural design of the implementation is explained. Section 5 presents some design and implementation details. Discussion and conclusion is presented in Section 7.

2. Tools and Technologies

The main software platforms for achieving the emulation system has been Blender 3D for modelling and its game engine as execution platform; the Ice™ communication middleware; and the Python interpreter. This section gives an brief introduction to these third-party software systems on which the current implementation rely.

2.1. *Implementation Overview*

The implementation of the systems for emulation and simulation is done entirely with Python[1]. The implementation is quite large and Python is a good choice as a code-efficient and intuitive language.

The separate systems that have been implemented and integrated as part of this work are[2]:

- *Math3D* (∼ 1000 lines of code) is a basic Euclidean mathematics library for working with positions, vectors, orientations, rotations, reference systems, homogeneous transforms, and linear interpolations.
- *PyMoCo* (∼ 2000 lines of code) is a motion controller framework implementing kinematics and various robot motion controllers, supporting real-time code and sensor interaction. The framework was developed and tested by Lind *et al.* (2010).
- *Emulation System* (∼ 2000 lines of code) is the code layer for all devices that can be part of a model of a production installation. For a given modelled device, a controller is implemented in the emulation layer, and the emulated controller must recognize the device configuration from the model. Any interactive device system is represented over the production control network by at least one Ice-server.

[1]`http://python.org`
[2]Generated using David A. Wheeler's 'SLOCCount'; confer `http://www.dwheeler.com/sloccount/`.

- *Production Control System* ($\sim$ 2000 lines of code) is the software layer of application control of the production installation; be it emulated or in reality. It is the objective of the production control system to operate the devices, cooperating among themselves.

2.2. *Middleware and Distributed Framework*

The choise of network communication and connectivity, i.e. the middleware, for a distributed control system is a central issue; especially so for one with real-time requirements.

The chosen middleware and distributed control framework was developed by Olivier Roulet-Dubonnet, and will be described in a later publication. The software is available for download and review from the *ColdHMS* project page on SourceForge[1]. The framework is called *IceHMS* in reference to the Ice™ (*Internet communication engine*) middleware and *Holonic Manufacturing Systems*. The reference to HMS is due to the original inspiration for the framework.

Ice™ is well motivated (Henning 2006, 2007), well described (Henning 2004), and well documented[2]. Good examples, performance towards the limit of no overhead[3], high availability, and clear semantics and syntax made Ice™ an favourable candidate.

A determining factor for choosing Ice™ is the cross-language and cross-platform properties. This will allow separate developer groups to stick with the platform and language they favour, or chose the platform and language best suited for their pertinent system implementation.

2.3. *Blender Game Engine*

Blender[4] is a (mesh) modelling and animation software studio. It comes with a quite advanced and efficient game engine, and integrates the Bullet physics library[5].

Through the embedded Python interpreter, the game engine is wide open for implementing real-time interaction of external control software with the internal game engine logic. The game physics may be utilized whenenver some mechanical part is under uncontrolled motion, like sliding or falling.

3. Demonstration System

The prototype production system, around which the demonstration emulation and simulation system is designed and implemented, is described by Lind *et al.* (2009). A preliminary status of the implementation was presented by Lind and Roulet-Dubonnet (2010).

The demonstrator is a realistic case of the logistics around production painting systems. The painting system consists of a chained trolley-conveyor, carrying workpiece carriers through the sites of the processes; such as washing, drying, painting, heating, and hardening. At certain areas along this main conveyor the up- and downloading of workpieces, and the occasional change of carriers take place. Workpieces are up- and downloaded

[1]`http://sourceforge.net/projects/coldhms/`
[2]`http://zeroc.com/download/Ice/3.4/Ice-3.4.1.pdf`
[3]`http://zeroc.com/articles/IcePerformanceWhitePaper.pdf`
[4]`http://blender.org`
[5]`http://bulletphysics.org`

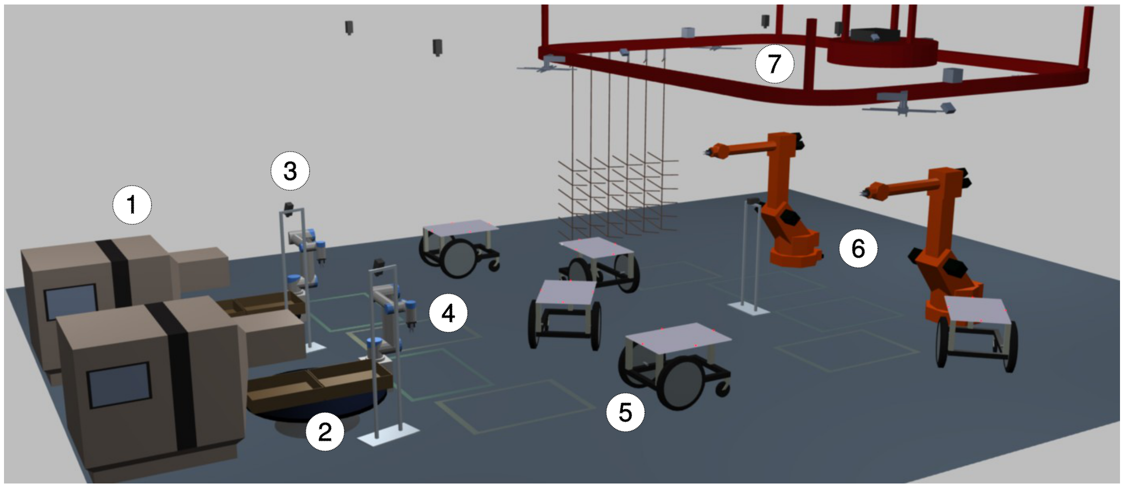


Figure 1. Rendered image from the Blender 3D model of the demonstration system.

for natural reasons, since every workpiece is to take only one tour of the painting system. Carriers are switched whenever a changeover rolls in a workpiece type with which the present carrier type is incompatible. The implemented demonstrator system only concerns uploading of workpieces.

The demonstration system is an extended version of the prototype system and is seen in Figure 1. In the order of flow of workpieces in the production system, the following gives a description of the mechanical devices and their operation.

(1) Workpiece machining, such as bending, turning, milling, extrusion, cutting, etc., is performed to single-piece order, in the CNC-machines at the far left.
(2) From a CNC-machine, workpieces drop from the outlet into a box fixed on a turntable. The turntables have two opposing boxes, one servicing a CNC-machine and the other servicing a robot.
(3) A vision system observes the box on the turntable-position that services the robot. As long as workpieces are localized in the box, the robot system can chose one and pick it.
(4) The robot opposing the CNC-machine at a turntable serves to arrange the workpieces into a specified layout on a docked AGV.
(5) The AGVs are made with a flat top-plate covered with a non-slip material suitable for workpieces to rest freely and steady onto during transport. They are able to dock into the faintly marked (yellow) buffer and (green) task areas on the floor, near the robots.
(6) On the far right, two robots are placed for handling workpieces off the AGVs and attaching them onto painting system carriers on the overhead Power-and-Free (PnF) conveyor.
(7) The PnF-conveyor is a decoupling installation from the main painting system conveyor; the latter is not modelled in the demonstration system. Empty carriers are transferred from the main conveyor to the PnF-conveyor and filled carriers the opposite way. A set of PnF-stops and associated presence-sensors enables the conveyor control system to manage the PnF-trolleys.

To give an impression of the resulting operation of the implemented demonstration system, some short video clips have been made. They may be downloaded in Xvid and Ogg Theora formats from `https://automatics.no-ip.org/~ml/intellifeed-emu/`.

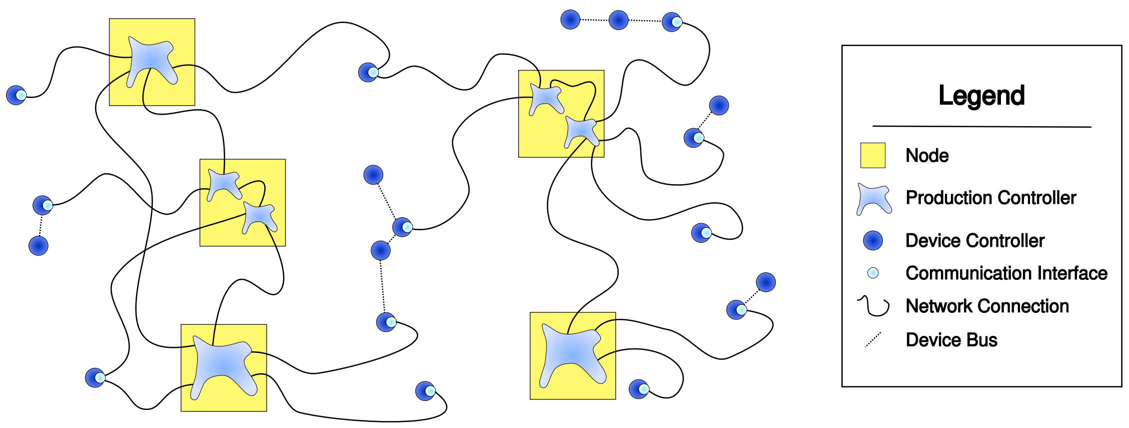


Figure 2. Example of connectivity in a distributed control system.

4. Emulation System Design

This section gives an overview of the architecture and design of the emulation system, and the technologies used for implementing it.

4.1. *Architectural Overview*

Conceptually the device controllers of the production system needs to be encapsulated in an emulation platform, and exposed appropriately to the production control system for simulation. An overview of the structure of a distributed control system, and how it is wrapped up for being emulated, is given here

4.1.1. Distributed Control Systems

Figure 2 shows an abstract view of a distributed production control system. The only concepts represented are control nodes, production controllers, device controllers, and connections. In addition there are two kinds of communications of different nature.

Most device controllers present a general communication interface, through which the production controllers may connect. The same kind of connection enables production controllers to interconnect for coordination and cooperation. Some device controllers are naturally distributed, and they communicate by some closed network or bus. This latter connection type is not of particular interest when developing a production control system, but this internal device controller network may be important when emulating the device.

There are no general constraints on the network connection topology. The specific types of production controllers to connect is naturally limited by semantics and types. This is determined by the setup in a production system, where the organization into factories, plants, lines, stations, and cells will play a large part in the particular connectivity and partitioning of the control nodes and production controllers.

4.1.2. Emulation and Simulation System

The objective of creating a system of emulated device controllers for the production controllers, shown in Figure 2, has been achieved with an architecture based on the Blender game engine. Figure 3 shows this architecture with the game engine residing on some central node. All device controllers are emulated in the embedded Python interpreter in the game engine. The embedded Python interpreter has access to the virtual reality of all the Blender objects modelled in the scene.

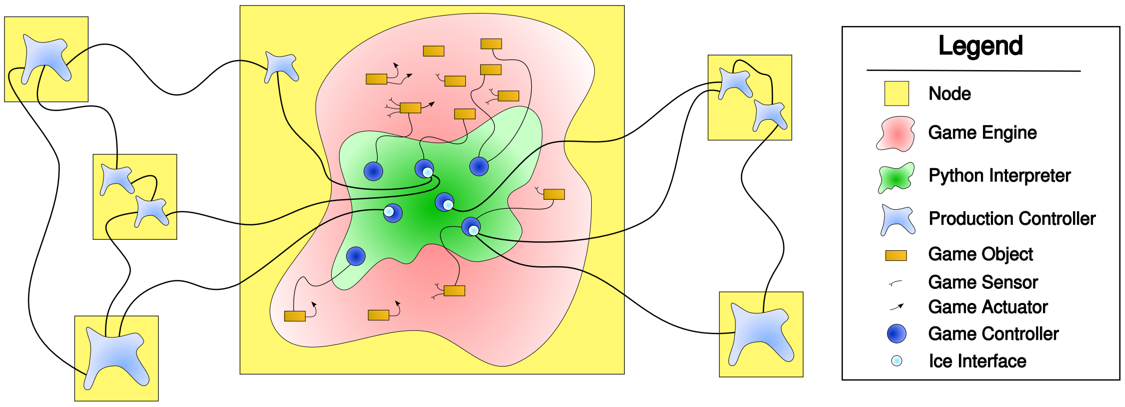


Figure 3. Overview of the architecture and connectivity in an emulated system.

Due to the versatility of the Python interpreter, the liberty given to it by the Blender game engine, and the cross-language capability of Ice™, it is possible to provide the same interfaces and functionality on the emulated device controllers, as on the real. Performance factors are described in the following:

- The frame-rate of the game engine decreases with increasing number and complexity of emulated device.
- Increasing the number of device controllers, or their communication load, may saturate the network bandwidth and latency capacities of the game engine node.
- Increasing the number of objects with enabled game physics decreases the frame-rate of the game engine.

These limitations on the game engine node capacities implies a bound to the production system complexity that will be feasible for emulation.

4.2. *Blender*

The Blender 3D application comprises modeller, animation engine, game engine. In the modeller, 3D geometric objects are drawn and annotated with information, configuration, and setup for animation, game execution, and physics. Game physics is a feature to be used in either animation or game mode.

4.2.1. Modelling

Modelling a production installation means making geometries representing visual objects, collision objects, physics objects, objects for mechanical sensors, etc., for the shop-floor scene. In addition, purely virtual objects must be modelled for the sake of emulating reality. Examples of the latter, purely virtual objects, are given here:

- For a vision system to localize certain workpiece types, a camera house object may be modelled for pure visualization. For the virtual vision system to operate an object is used to represent the frustum. Thus, the operation of the vision system is to find all object of the requested workpiece type within the frustum. Tests for visibility from the camera is done by ray-casting from the apex of the frustum to filter out occluded workpieces.
- A proximity sensor can be modelled by adding an invisible object, with an associated collision sensor, the mesh of which spans the region of sensing. If the sensing should be selective on material properties, like capacitive or inductive sensors, the objects to

sense should be marked with a material property to be distinguishable.

4.2.2. Game Engine

The game engine, once started, turns every model object of the current scene into a game object. A game object may contain any number of game sensors, game controllers, and game actuators. These are pure Blender game engine concepts, unrelated to production devices.

The game sensors of various types relate the associated game object to user input, message reception, collisions, timing events, ray tracing, and actuator events. The game actuators affect objects in the scene, like moving, creating, destroying, changing properties, etc. The simple game controllers are Boolean operators, which test their connected game-sensors and translates the resulting truth value to triggering of the connected game actuators.

For simple usage of the game engine the Boolean controllers may be adequate. However, for implementing advanced behaviour, the game controller can be chosen to be a callable object in a Python module. This type of controller will, when triggered by any sensor, execute the callable Python object in the Python interpreter embedded in the game engine. There is no limit to what a Python callable is allowed to access, in the game engine as well as with the general operating system environment.

A game controller on some object can access all state information in all of the sensors, regardless of how it was triggered. For instance, a game sensor set up for triggering its game controllers on some event in the game engine is an efficient event-based method for activating the controllers. Functionally the same behaviour may be achieved by polling of the sensor by the interested controllers. This can be more flexible, whenever the controller has multiple concerns, not all of which are related to the mentioned sensor.

4.2.3. Game Physics

Game physics gives realistic dynamic behaviour for objects under its control. It gives a generic method to emulate reality for objects which are not under explicit motion control in reality. I.e. objects that in the real world have no controller or mechanical manipulator. There are plenty of examples thereof, like passive joints and links in robots; workpieces falling from machines, grippers, or conveyors; obstacles being bumped into by AGVs; textile or other flexible material which is only partially fixed; etc.

One strategy for such free objects in the emulated production system is to explicitly control them. I.e. to implement controllers that execute the behaviour of the free objects. This requires a great deal of object-specific knowledge accessible to the controller.

Consider the example of machined workpieces dropping out of a machining station. Such workpieces may have highly complex geometries, and they may even be a one-shot batch. The stable stances of such workpieces on a flat support may be easily enumerated. The transitions from the dropping phase-space state (trajectory) to the final stance may be somewhat more complicated, even when landing on a flat surface; at this point friction and elasticity, or even plasticity, come into play. Finally, since there will typically be several workpiece in the container under the machine outlet, most of the workpieces drop on top of some structure of other workpieces, scattering and bouncing before settling in the structure. Generating offline-knowledge for the controller to use for generating realistic scenarios in real-time for such, typical situations seems impossible in the general case. If possible, the solution will be very unrealistic; the knowledge of the controller will be more a kind of simulation scenario, specific not only for the workpieces but also for the environment; or there are some very simplifying features or assumptions of the specific workpieces and the specific container.

The game physics in Blender gives the opportunity to undertake an alternative strategy. By modelling a collision body as a compound of convex rigid bodies, alongside the visible, geometric skin of the workpiece, it is possible to achieve physically realistic dynamic behaviour of the workpieces with their environment and among them.

There are some drawbacks with using game physics.

- The Bullet physics library does not feature static friction. This may not seem severe at first, but it implies unrealistic scenarios for parts lying at rest on a support. If the support accelerates however weakly horizontally or has even the slightest inclination, the supported parts slide on the surface.
- The physics computation, even for a moderate amount of bodies, is quite heavy; and it must be kept in mind that this computational load can currently not be distributed out of the game engine process. To achieve acceptable performance for a complex production scenario with physics based dynamics, high-performance computing hardware may be necessary.

The two drawbacks of using game physics can be resolved by the same solution. The solution is based on separating the physics dynamics body from the workpiece geometry and appearance, and to be able to *freeze* the workpiece geometry to where it settles, freeing up the physics dynamics body for use by other workpiece geometries. By maintaining a pool of and reservation system for physics dynamics bodies in the game engine, only a limited number is necessary, and thus limits the load on the CPU. Simultaneously it resolves the problem with slipping workpieces, since they will freeze onto their supporting surface when below some motion threshold.

5. Device Emulator Designs

This section describes and discusses some aspects of lower level (mechanistic) design, towards implementation.

The controllers of devices in the real production system are the targets for implementation of device emulators. A device emulator interacts with the game model, i.e. its own device model, other device emulators, and other modelled elements, in much the same way as the real device interacts with its surroundings in reality.

A device is modelled geometrically with Blender, and annotated appropriately for functionality. The device control emulator is then implemented in Python and associated with the geometrical models in the game engine.

5.1. *Implicitly Controlled Devices*

The virtual production reality in the game engine contains activities for which there are no explicit controllers in the real production system. Such natural activity in reality may play an implicit role in the design of the production control. An example is that workpieces released from a gripper will drop downwards, and it will come to rest if supported by a level surface or a fixture. This mechanism is implicitly used in most pick-and-place systems in production automation.

As suggested in Section 4.2.3, and depending on the implicated complexity, the strategy of resolving the dynamic control of free objects may, or may not, involve game physics. Thus, either the controllers of the solution takes on the explicit dynamics of the emulated

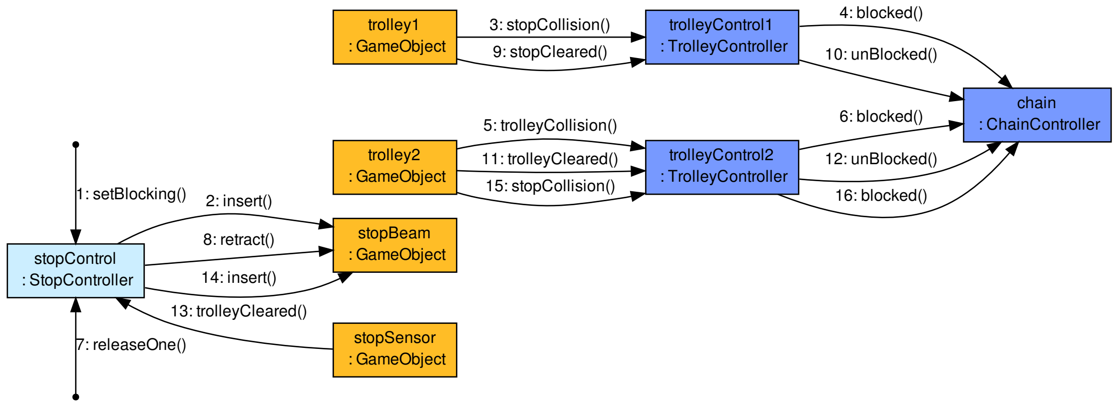


Figure 4. Communication diagram for the simple scenario of two trolleys approaching a PnF-stop. Exposed game engine controllers are shown in light blue, unexposed controllers in blue, and game engine objects in orange.

subsystem in its entirety, or the dynamics of the subsystem objects are left entirely to game physics. As it turns out, a hybrid solution is possible, where a set of controllers manages the deployment and activation-states of physics enabled devices.

In the following, two mechanisms for handling uncontrolled objects will be exemplified. One deals with trolleys on the PnF-conveyor, using direct, unexposed controllers for driving the trolleys and interacting with other trolleys and PnF-stops. The exact same mechanism is applicable on pallet conveyors. The other example is game physics control of workpiece objects.

5.1.1. PnF-Trolleys

The dynamics of the PnF-trolleys in the PnF-conveyor of the demonstrator system, the elevated red track in Figure 1, are handled by direct, unexposed control. The control of the trolleys are unexposed to the production control system, and simply realizes the physical constraints under which they move on the conveyor.

Trolley controllers move their game objects upon reception of motion updates from the conveyor chain controller. In case a trolley is blocked, it unsubscribes from such motion updates. Trolleys can be physically blocked by either bumping into the blocking-beam of a PnF-stop, or by bumping into the rear of other blocked trolleys. Either situation generates a collision event from the trolley game object, and the trolley controller unsubscribes from the motion updates. Whenever the interference from the preceding collision is cleared, the trolley controller receives another collision event from the trolley game object, and it resubscribes to motion updates.

The motion updates, tied to the frame-rate of the game engine, from the chain drive gives enough information to the trolley controller, that it can update its position along the conveyor track. The conveyor track is simply modelled as a closed polygon-mesh in Blender. In the demonstrator, the conveyor is a single circular loop, but for more advanced overhead- or pallet-conveyors, branches in the track may occur, and diversion-controllers must be implemented for the intersections on the paths.

Figure 4 is a communication diagram for controllers and game objects in a scenario involving one PnF-stop and two PnF-trolleys. The scenario assumes initially that two trolleys are approaching the PnF-stop.

(1) The PnF-stop is ordered, by external activation from the production control, into the blocking state.

(2) The stop controller inserts its stop beam into the track.
(3) As the first trolley interferes with the beam, its collision sensor triggers the trolley controller.
(4) The trolley controller unsubscribes from motion updates from the conveyor chain controller.
(5) As the second trolley gets into collision with the first trolley, its collision sensor triggers.
(6) The second trolley controller unsubscribes from motion updates from the chain controller. The state is now that both trolleys are blocked at the PnF-stop.
(7) The external production control orders the PnF-stop controller to release one trolley.
(8) The PnF-stop controller orders the retraction of the beam from the track.
(9) This collision sensor of the first trolley triggers its controller, signalling that the stop beam interference has been cleared.
(10) The trolley controller resubscribes to motion updates from the chain drive controller, and it starts moving the trolley again.
(11) As the first trolley moves out of interference with the second trolley, the controller for the second trolley gets triggered, signalling the cleared interference.
(12) The second trolley controller resubscribes for motion updates.
(13) The presence sensor at the PnF-stop now triggers as the first trolley completely passes the PnF-stop.
(14) The PnF-stop controller reacts to this by re-inserting the beam into the track.
(15) As the second trolley object reaches the beam, its collision sensor triggers the controller.
(16) The second trolley controller unsubscribes for motion updates.

Thus the final state is that the first trolley is moving along downstream of the PnF-stop, while the second trolley remains blocked at the PnF-stop.

The dynamics of the trolleys could have been designed by using constrained motion of physics objects, and would then have been even more realistic. However, the chosen design of direct control is sufficiently generic and less CPU-intensive.

5.1.2. Workpieces

The simplicity of the control logic and emulation of trolley- and pallet-conveyors, being fixed installations within one production application and varying little across applications, favours the direct control strategy. Workpiece types and environments, on the other hand, may exhibit high variation even within a single production application in flexible manufacturing. If opting for the direct control approach, as mentioned in Section 4.2.3, the designers and implementers of the emulation system faces high complexity of knowledge management regarding stable states, mechanical interactions, and dynamic control of the workpieces.

For these reasons, the game physics approach is a natural choice for workpiece management. To limit the computational load, the hybrid approach was chosen. The technique is to manage a limited pool of dynamic bodies, where attachment and control of a workpiece skin is done on request.

The following describes a scenario where a new workpiece is requested to enter the production scene. In the production demonstrator, cf. Figure 1, this occurs at the CNC-machines on the far left.

The scenario in Figure 5 involves the singleton objects for workpiece management and workpiece body pool. A workpiece body object is a compound of physics enabled objects,

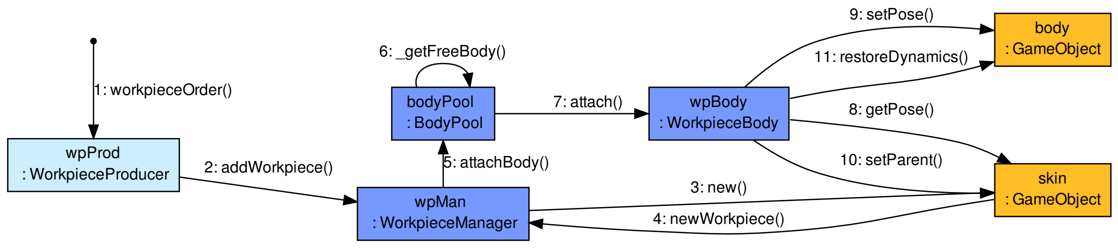


Figure 5. Communication diagram for entering a new workpiece into the emulation.

and is owned by a workpiece body controller. The body controller handles the logic of suspending and resuming dynamics for its body object, as well as placing the body at, and attaching the workpiece skin to it. For every site in the production scene where workpieces can be generated, there exist a workpiece producer. The workpiece producer initiates the whole process of creating and handling a newly created workpiece skin, as the following description illustrates.

(1) From the external production control, an order to create a new workpiece arrives at the workpiece producer.
(2) The workpiece manager is instructed to create a new skin.
(3) A new skin object comes to existence in the game engine.
(4) Any new skin starts its life cycle by informing the workpiece manager that it has come into existence.
(5) The body pool is instructed by the workpiece manager to attach a body of the correct type for handling the dynamics of the free skin.
(6) The body pool internally allocates a free body controller of the appropriate type.
(7) The body controller is handed a reference to the pertinent skin.
(8) The pose of the given skin is retrieved.
(9) The body object is set in pose to coincide with the skin.
(10) The skin is parented to the body, which effectively makes it move with the body.
(11) Finally the physic dynamics of the body object is reestablished, making it interact in a mechanically realistic manner with its surroundings.

The implemented logic that takes over after the described scenario, in the workpiece body control, is to first test whether the body is released in collision. If so, the dynamics is immediately suspended, the workpiece skin is frozen onto one of the objects it interferes with, and the body is freed at once. This behaviour is deliberately implemented for easy emulation of an attachment process. On the other hand, if the body is not initially in collision, dynamics ensures that the workpiece will start falling, tumbling, sliding, as long as the amount of motion exceeds a certain threshold. When the amount of motion goes below the threshold, the same method of freezing the workpiece skin is employed, as if there was an initial collision.

5.2. *AGV Control*

The devices internal to a minimally modelled AGV are a velocity controller and a localizer. These devices are easily emulated in the game engine. The velocity controller presents an interface with a single method for setting a 2D velocity screw, composed of two linear and one angular velocity components. At every time frame, the position and orientation of the AGV object is displaced according to the 2D velocity screw. The localizer presents an interface of one method for retrieving the world pose of the AGV, which is read out as the pose of a central geometry of the AGV in the game engine.

These basic devices are realistically emulated in terms of interface, but too ideal in terms of precision and stability. A real velocity controller runs on data from odometers and inertial sensors, the precision and stability of which may be far from ideal. The real external positioning system uses vision-systems for recognition and tracking of LEDs on the AGVs, for which there may be abundant calibration and stability issues. Thus, it is mostly the interface and the nature of the data from the emulated devices that are realistic, rather than the detailed behaviour.

Based on these basic emulated devices, the AGV controller itself, composed of more abstract controllers like motion control, tasking control, traffic and navigation management, etc., may be implemented outside of the game engine. This is an example of crossing the game engine boundary for implementing the devices, and will help distribute the computational load on the game engine node in the system of simulation nodes. In a real application, this implies that the whole AGV controller system, except for the fundamental velocity controller and localizer, can be ported directly from the simulation to the real AGV platforms.

It is worth noting that even the velocity controller could be decomposed and moved to an external node. This requires leaving the set of motor controllers, and the odometers as the elementary devices to be emulated. Though this design may eventually alleviate further the computational load on the game engine node, the frequency of communication among velocity controller, odometers, and motor controllers may be so high that the communication load on the game engine node becomes the limiting factor for performance. This may not only be the case for the emulation system, but may also be true for the real AGVs.

5.3. *Robot Control*

The robot controllers and kinematics are based on a framework called *PyMoCo* (Python Motion Control). For a general description of PyMoCo, confer Lind *et al.* (2010), and for a demonstration in application, confer Schrimpf *et al.* (2010). Motion control in the PyMoCo framework is based on UDP-socket interaction with the native robot low-level controller.

Each robot low-level emulator in the emulation system initializes by finding the game engine objects belonging to its particular robot, gets allocated a socket port pair, and starts up listening for control input on the game engine node host. At some node in the production control, a program can be started for setting up a motion controller with PyMoCo, or any other motion control method, and provide a motion control interface over Ice™ to the production control system at large.

The implemented design for robot control in the emulation system is shown in Figure 6. At the highest level, the *Task Handler* receives macroscopic motion commands, which amounts to moving the robot tool frame linearly to a given 3D-pose. The *Task Handler* sets up the *Task Space Interpolator* for the *Joint Data Controller* to use. The *Joint Data Controller* requests task space poses from the *Task Space Interpolator* in real time, and uses the *Inverse Jacobian* to compute the corresponding joint motion to achieve the interpolated task space pose. The computed joint target is sent to the *Low-Level Controller*, which, as a game engine controller, resides inside the game engine. In case of the real low-level controller, the joint targets are adjusted and immediately distributed to the individual servos of the robot arm. In the emulated robot device, the *Forward Kinematics*

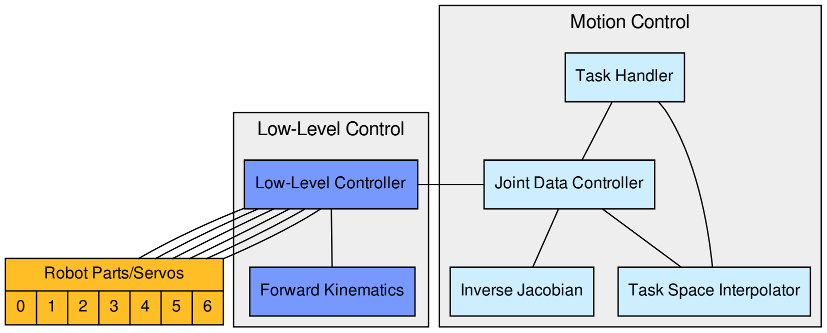


Figure 6. Object diagram for the implemented design for motion control.

is used to compute where to place the geometric objects, representing the parts of the robot, according to the given joint target.

The discussion pertaining to deployment and distribution of control elements for AGVs applies equally to the robot emulation; confer Section 5.2. It is possible, but with high computational load ensuing, to implement the motion control inside the game engine. This computational load of the game engine node is traded off for higher communication load, by having the motion control outside of the game engine. In effect, this leaves the low-level control and the associated computation of kinematics in every time frame inside the game engine. The separation at this level, i.e. low-level control inside and motion control outside of the game engine, is natural since that separation exists in real robot controllers. However, at no considerable extra communication overhead, the forward kinematics computation could be moved outside the game engine by simply sending poses of robot parts rather than joint data over the low-level controller input socket. In the current implementation this has not been considered due to the implied violation of the philosophy of having raw devices truly emulated in the game engine.

6. Performance Indication Experiments

As touched upon in Sections 4.1.2, 4.2.3, and 5.2, a number of parameters and circumstances affect the performance of an emulation setup, and thereby the entire surrounding simulation. Whether an obtained actual performance is acceptable for a given emulation and simulation setup will depend on the real-time nature of the pertinent application.

Absolute performance of a system setup executing with the Blender game engine is heavily depending on the quality and performance of many parts of the computer system setup as a whole. As is often the case in such situations, it is better to focus on the relative behaviour of certain interesting or indicating observable quantities as a function of problem size; conveying an impression of the performance *scalability* of the principles and technologies.

6.1. *Test Considerations*

With the possible exception of physics-enabled, colliding objects, it is clearly most interesting to make performance test of scalability on independent devices. The performance observables will be presented as amortized values, to clearly emphasize deviations from linear scaling. Complexity arising from interaction between controllers of emulated production devices is not to be attributed to the emulation system, but rather to the simulated production control system.

Real-time emulation clearly distinguishes itself from discrete event simulation by not controlling real time. The factors that may be controlled inter-dependently in real-time emulation, and are critical indicators for performance, are CPU-load and frame-rate[1] The Blender Game Engine has a switch called *"Enable All Frames"*, which when set drives the emulation in a "free-running mode". When set, the frame-rate is in principle unlimited, and determined by the system resources such as performance and availability of network, CPU, GPU, RAM, bus speeds, etc. Depending on the nature of the executing emulation together with all other resource loads on the, possibly distributed, computing system, one of these will, at any given time, be the limiting factor on the achieved frame-rate. If *"Enable All Frames"* is unset, a parameter in the game engine sets a fixed upper limit of the frame-rate. Naturally, any of the aforementioned system resources may be fully loaded at a lower frame-rate; which then becomes the actual frame-rate in the emulation.

This leads to two pure performance indication principles.

- **CPU Consumption:** With *"Enable All Frames"* unset and a fixed target frame-rate, the CPU consumption for a set of tasks, of a given duration, may be considered a performance indicator of the game engine among the tasks. The CPU consumption is measured by reading the process resource statistics from the operating system before and after the task, and is taken as the sum of both system and user times consumed for the task period. Obviously, a lower CPU consumption is considered better performance.
- **Achieved Frame-Rate:** With *"Enable All Frames"* set and continuous monitoring of frame-rate and task specific state or quantity, these may be correlated to give a performance indication. A higher frame-rate indicates better performance. Since frame-rate is not an easy quantity to amortize over a problem size value, a more appropriate measure is the reciprocal of the frame-rate; the *emulation cycle time.*

Both of the CPU consumption and the achieved cycle time are easy to test for scalability by amortizing them over the problem size.

6.2. *Experiment Setups*

Three production-relevant experiment setups have been used for performance-testing the use of the Blender game engine. Two of these setups are presented in the following sections and they are used to test directly the scalability in terms of number of devices and physics-enabled objects in the game engine.

The experiment setup that is not presented in detail here was set up for running a variable number of emulated robot controllers. The robots were controlled from a remote PC, over a switched $100Mb/s$ network, in one single process running the corresponding motion controllers. The task for each motion controller was to drive its robot through a fixed number of pick-and-place cycles and with all tasks running simultaneously. Such a setup characterizes a performance test of the motion control framework set up on the motion control PC, rather than a performance test of the game engine setup. The motion control PC managed to control up to ten robots before being too loaded to respond in a reasonable real-time manner; i.e. when the motion of the robots in the game engine scene started to exhibit an observably jagged motion. Up to this maximum of robots, only a

[1]The frame-rate is the frequency with which the scene is displayed and the maximum triggering-rate of the game sensors.

very slight increase in amortized CPU consumption was observed on the motion control PC. It has not been investigated if the total CPU load on either the motion control PC or the game engine PC was the limiting factor. This will be an objective for a future experiment.

The remaining two experiment setups that were isolated to, and targeted the internals of the game engine, were both deployed on four different hardware setups with near-identical systems environments. All four systems used the Debian GNU/Linux "testing" distribution and the same versions of Blender, Python, ZeroC Ice$^{\text{TM}}$, etc. The configurations of the systems environments were left very close to the defaults for a freshly installed Debian GNU/Linux system. This is not by any means a guarantee that the performance test results may be used for comparing the four hardware platforms; since the installation process performs a lot of auto-detection and auto-configuration, and because drivers for different hardware peripherals may exploit or disregard a multitude of features. However, it does indicate what is to be expected out-of-the-box and it makes it quite easy to repeat and verify the results.

The four hardware platforms consisted of two desktop PCs and two laptops; all with enough RAM that no paging occurred during any experiment. The rough platform specifications are given here with a mnemonic label for each hardware system:

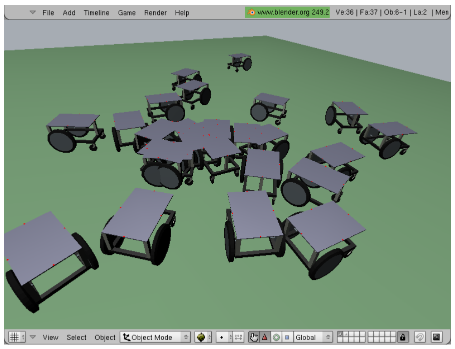
- **i7-860+gtx260** A high-end desktop with the Intel i7-860 processor and a graphics card based on the NVidia GTX260 GPU. This is the absolute top line of the four systems in any respect.
- **d525+gt218** A very low-end mini-desktop PC, with the Intel Atom D525 processor and an integrated NVidia GT218 GPU. Though the GPU is fair, the processor makes this the system with the lowest expectations,
- **t8300+gm965** A Lenovo Thinkpad X61 laptop with the Intel Core 2 Duo T8300 processor and the integrated GMA X3100 GPU in the Intel GM965 chipset.
- **t7300+gm965** The same configuration as the Thinkpad X61, but with a lower performing Intel Core 2 Duo T7300 processor. The laptop is a Lenovo 3000 V200.

All experiments presented in the following sections are based on one single run of the setup for each system, with *"Enable All Frames"* set and continuous logging of the number of devices or objects together with the actual frame-rate. Statistics is generated by steadily increasing the number of objects while ensuring that an as compact as possible coverage of the achievable domain is sampled.

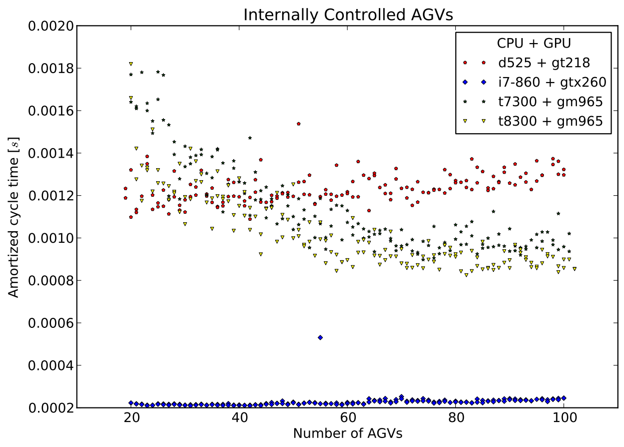
6.3. *Experiment Setup: Internally Controlled AGVs*

The most central question for any production emulation scenario is how the performance scales with the number of active, production related, controlled devices in the game engine. An emulation experiment has been set up, where the number of emulated AGVs is automatically and continuously increased with regular intervals. The experiment is carried out in one single emulation run, with frequent logging of the number of AGVs and measured frame-rate. The setup is displayed in Figure 7(a).

The measurements were made in the range of 1 to 100 AGVs, and the results are displayed in Figure 7(b) as amortized cycle time against number of AGVs. In order to increase identifiability of the higher-count end of the measurements, the lower, peaking tail at the low-count end has been truncated at approximately 20 AGVs. The reason that the amortized cycle time peaks at the low-count end is that few devices each gets a considerable contribution in the amortization of the constant load of the game engine;



(a) AGV performance experiment scene.



(b) Amortized cycle time over number of AGVs.

Figure 7. Game engine setup and result for performance of internally controlled AGVs.

i.e. the part of the game engine load not associated with AGV control.

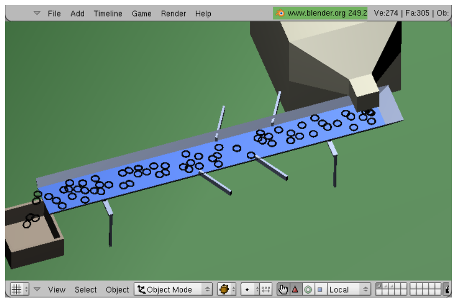
It is not surprising that the high-end desktop PC, i7-860+gtx260, is far superior to the other systems. The desktop systems, i7-860+gtx260 and d525+gt218, exhibit an expected behaviour with a slight super-linear computation time, i.e. slightly increasing amortized cycle time. It is, however, surprising that the two laptop systems, t8300+gm965 and t7300+gm965, exhibit a slight decrease in amortized cycle time, i.e. sub-linear increase in computation time.

6.4. *Experiment Setup: Physics-Enabled Objects*

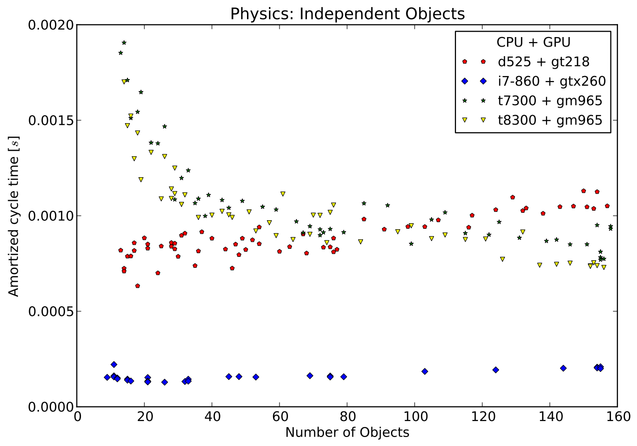
Though physics-enabled objects, categorized as “Dynamic Body”, “Rigid Body”, and “Soft Body” in game engine terms, are not strictly necessary for most production system emulations, it may be an important contribution to the realism of the emulation, and in many cases it directly alleviates the need for implementing an artificial virtual-physical workpiece management system. In certain production settings, or aspects thereof, the physical properties of bouncing, tumbling, slipping, and sliding are of paramount importance to the usefulness of an emulation of the production system. This section presents an experiment setup and two experiments for performance testing of physics-enabled objects. This may be characterized not as an isolated performance test of the Blender game engine, but rather as a combined test with the integrated Bullet physics library.

A device type that is found abundantly in production settings due to its low cost, flexibility, and reliability, is the *Vibrating Feeder*. It is well suited for performance testing of physics-enabled objects, because it bases its operation massively on physics properties like friction and elastic collisions, involving many interacting or separated workpieces.

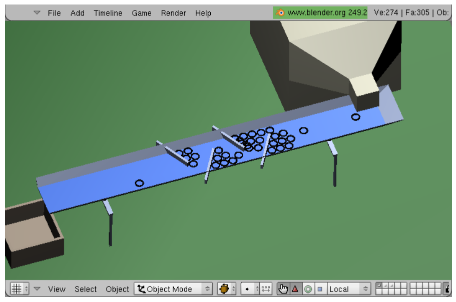
A simple model of a vibrating feeder is made up by a surface of friction bounded on the sides by low-friction inclined plates. By simply toggling the feeder plate between two narrowly displaced positions in any game engine time frame, objects will vibrate or bounce, depending on the size and direction of the displacement. Many aspects and parameters are possible to model, such as frequency, number of positions and the pattern of motion among them. Such effects and parameters will affect the motion of the supported objects, very much so in the game engine as in a real vibrating feeder. For the purpose



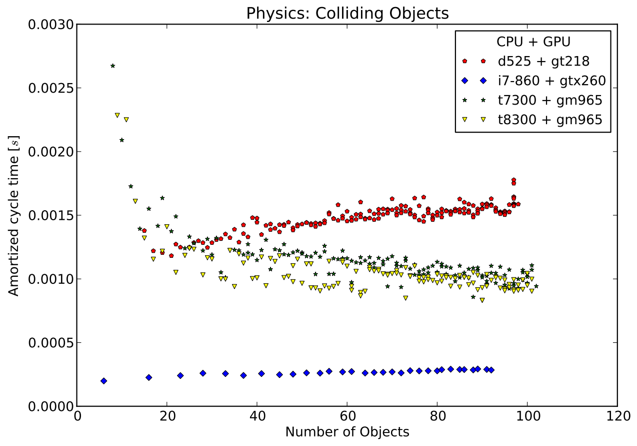
(a) Physics objects performance experiment scene for limited inter-object collisions.



(b) Amortized cycle time over number of objects for light (*rate controlled*) inter-object collisions..



(c) Physics objects performance experiment scene for heavy inter-object collisions.



(d) Amortized cycle time over number of objects for heavy (*barrier controlled*) inter-object collisions..

Figure 8. Game engine setup and results for performance of physics-enabled objects. The two cases refer to controlling the number of physics-enabled objects, O-rings on a vibrating feeder, by either putting barriers into the flow of O-rings or increasing the production rate of O-rings.

of the experiments presented here, it suffices that an adequate direction and length of displacement may be configured in the local reference system of the feeder surface.

There are two experiments performed in the vibrating feeder setup. The first experiment sets up the objects for being fed separately along the feeder surface, in principle only interacting with the feeder surface; the ***Free Flow*** experiment. This setup is shown in the screen-dump in Figure 8(a). The second experiment provokes massive inter-object collisions in addition to the interaction of each object with the feeder surface; the ***Barrier*** experiment. This is obtained with the same setup, but with a set of barriers configured to be slid into the flow of objects, creating pockets where objects are trapped. The experiment setup is seen with the barriers shifted into the object-flow in Figure 8(c). To the far right in the scenes is seen an emulated machine, that produce workpieces[1] at a dynamically controlled rate. The O-rings drop from the machine outlet onto the vibrating surface, being transported along to the far left where they drop freely over the end of the feeder into a box where they are consumed; and destroyed as game engine objects.

The free flow experiment is carried out by a continuous increase of the steady state number of objects on the feeder, by manually adjusting the production rate of the O-ring producer. The process is allowed to run from a few objects and until the frame-rate drops to a few Hertz. The barrier experiment runs automatically with constant production rate of O-rings, and the barrier pockets fill up and trickle over one by one. As the last barrier overflows, and objects begin to fall off the feeder, the number of objects enters a steady state and the experiment is stopped.

The two experiments are highly relevant to real application scenarios of production, where transport and dynamic storage of small workpieces are frequent uses of vibrating feeders.

Results for the experiments are seen in Figures 8(b) and 8(d). The results exhibit much the same characteristics as was the case for the experiment setup with AGVs and the same comments and conclusions apply here.

7. Discussion and Conclusion

The game engine for use in real-time emulation and simulation of production systems is an example of *preadaptation*; a term borrowed from evolutionary biology. The designers and developers of the Blender game engine did not consider the domain of production control simulation. However, as there is a large overlap in geometric, logic, and real-time concerns of a 3D computer-game and production device emulation, the game engine platform turns out to nicely support this domain.

The proposed principle of a clean separation of device-emulation and production-control is considered a major contribution of this paper. The level of complexity of the demonstrator and implementation is almost complete with regard to architecture and design. There seems to be no hindrances for expanding to the complexity-level of the real production system; i.e. scenarios involving changeovers where painting system carriers, robot grippers, and workpiece types are changing during operation. Specifically, a few extra, uncomplicated devices will have to be added to the emulation system, whereas the real complexity of the full-scale scenario will appear at the production-control level.

The currently implemented production control system for testing the emulation system follows the operations described in Section 3. It does not perform any kind of scheduling and there exist only few instances of device reservation. These are central concepts for a production system that support device and machine utilization across several coincident product-setups over the same production resources. In future work, the production-control system will be targeted for full implementation, and the emulation system will be of central value for that development.

While the implemented AGV-system serves the purpose for logistics operation in the demonstration, it is much too simplified for realistic emulated operation. The value of the current implementation of the emulation system is also apparent here, used as development and test platform for implementing missing functionalities; e.g. trajectory planning and control, external localization system, and obstacle identification and avoidance.

[1]The workpiece used is an O-ring modelled by a torus with approximately 1600 polygon faces.

The implemented demonstration runs at several tens of Hertz on a modest, contemporary PC. While a severely wrongly balanced load of computation may impair the frame-rate to the level of being unusable for robot control or interference detection, this is easily avoidable. The ever increasing power of common PCs together with game engines taking advantage of new and emerging parallel processor architectures, e.g. GPUs, will enable acceptable performance of real-time emulation of entire flexible factories.

Production system engineers may take advantage of an emulation-setup, such as the demonstrator presented in this paper, as a training platform for conceptual design, implementation, commissioning, and run-in of a new production control system. Remaining issues during real commissioning may even have been foreseen during emulation-training; which is highly preferable to dealing with unknown issues during expensive downtime of a production installation.

Three experiments have been carried out and presented to address the performance of the emulation execution in the game engine; confer Section 6. Each experiment was carried out on four different computers, comprising two desktop PCs and two laptops. All experiments exhibit a reassuring, near-linear performance complexity. A consistent slightly sub-linear complexity in all experiments on the laptop platforms is the most surprising result from the experiments. This phenomenon has not been explained or investigated. There are two immediately plausible explanations, which will be investigated in the future:

- Even though the operating system and run-time environments are close to identical on all four computer platforms, the laptops are identified automatically as targets for power saving. Thus, with increased load on the system resources during an experiment, the number of active power saving features decrease, and the system performance increases.
- The GPUs of the laptop systems are inferior to those of the desktop systems, and they are the limiting factor at low problem sizes where the game engine runs with a high frame-rate. As the problem size increase, the CPUs become more loaded and they eventually become the limiting factor at the higher problem sizes.

Both explanations are consistent with poor performance at low problem sizes and with the slightly sub-linear overall behaviour. The test results lead to the overall conclusion that there are, within the domains tested, no super-linear effects that hampers the scalability of performance with the problem size in an emulation.

The suitability of the developed principles, architecture and chosen technologies, has been successfully demonstrated by implementation for a medium sized section of a production system. Hence, we conclude that our approach is applicable as a development support within the domain of distributed, autonomous production control.

Acknowledgements

The authors wishes to thank The Norwegian Research Council for funding this work through the IntelliFeed research project. Thanks also go to the industrial project partners Glen Dimplex Nordic AS and Scandinavian Business Seating AS for good input and interaction. Finally, thanks to Production Technology, SINTEF Raufoss Manufacturing AS for good support throughout the project.

References

Caie, J., 2008. *Discrete Manufacturers Driving Results with DELMIA V5 Automation Platform* [online]. : ARC Advisory Group. ARC White Paper on behalf of Dassault Systèmes [2011-04-01].

Hall, K.H., Staron, R.J., and Vrba, P., 2005. Experience with Holonic and Agent-Based Control Systems and Their Adoption by Industry. *In*: V. Marík, R.W. Brennan and M. Pěchouček, eds. *HoloMAS*, Vol. 3593 of *Lecture Notes in Computer Science* Springer, 1–10.

Henning, M., 2004. A New Approach To Object-Oriented Middleware. *IEEE Internet Computing*, 8 (1), 66–75.

Henning, M., 2006. The Rise and Fall of CORBA. *ACM Queue*, 4 (5), 28–34.

Henning, M., 2007. API Design Matters. *ACM Queue*, 5 (4), 24–36.

Kim, H., Zhou, C., and Du, H.X., 2000. Virtual Machines for Message Based, Real-Time and Interactive Simulation. *In*: *Winter Simulation Conference*, Vol. 2, Orlando, Florida, USA San Diego, California, USA: Society for Computer Simulation International, 1529–1532.

Lind, M. and Roulet-Dubonnet, O., 2010. Emulation of Manufacturing Devices for Simulation of Distributed Real-Time Control. *In*: T.K. Lien, ed. *Proceedings of the 3rd CIRP Conference on Assembly Technologies and Systems*, Trondheim, Norway NO-7005, Trondheim, Norway: Tapir Academic Press, 67–72.

Lind, M., *et al.*, 2009. Holonic Manufacturing Paint Shop. *In*: V. Marík, T. Strasser and A. Zoitl, eds. *Holonic and Multi-Agent Systems for Manufacturing*, Vol. 5696 of *Lecture Notes in Computer Science* Springer Berlin / Heidelberg, 203–214.

Lind, M., Schrimpf, J., and Ulleberg, T., 2010. Open Real-Time Robot Controller Framework. *In*: T.K. Lien, ed. *Proceedings of the 3rd CIRP Conference on Assembly Technologies and Systems*, Trondheim, Norway NO-7005, Trondheim, Norway: Tapir Academic Press, 13–18.

Marík, V., Vrba, P., and Fletcher, M., 2005. Agent-Based Simulation: MAST Case Study. *Emerging Solutions for Future Manufacturing Systems*, 159, 61–72.

Moon, D.H., *et al.*, 2006. A case study of the body shop design in an automotive factory using 3D simulation. *International Journal of Production Research*, 44 (18–19), 4121–4135.

Pannequin, R. and Thomas, A., 2010. Emulica: an emulation-based benchmarking framework for production control experiments. *In*: *10th IFAC Workshop on Intelligent Manufacturing Systems*, Lisbon, Portugal, Jul.. IFAC.

Park, C.M., Park, S., and Wang, G.N., 2009. Control logic verification for an automotive body assembly line using simulation. *International Journal of Production Research*, 47 (24), 6835–6853.

Schrimpf, J., *et al.*, 2010. Real-Time Sensor Servoing using Line-of-Sight Path Generation and Tool Orientation Control. *In*: T.K. Lien, ed. *Proceedings of the 3rd CIRP Conference on Assembly Technologies and Systems*, Trondheim, Norway NO-7005, Trondheim, Norway: Tapir Academic Press, 19–23.

Smith, J.S., 2003. Survey on the Use of Simulation for Manufacturing System Design and Operation. *Journal of Manufacturing Systems*, 22 (2), 157–171.

Valckenaers, P. and Van Brussel, H., 2005. Holonic Manufacturing Execution Systems. *CIRP Annals - Manufacturing Technology*, 54 (1), 427–432.

Vrba, P. and Marík, V., 2005a. From Holonic Control to Virtual Enterprises: The Multi-Agent Approach. *In*: R. Zurawski, ed. *The Industrial Information Technology Hand-*

book. CRC Press.

Vrba, P. and Marík, V., 2005b. Simulation in Agent-based Manufacturing Control Systems. *In*: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Vol. 2, Oct.. IEEE, 1718–1723.