# Using Technology in the Computer Science Classroom: Using a Scriptable Game-engine for Teaching Artificial Intelligence

**Malan den Heijer**

**Roelien Goede**

*North-West University, Faculty of Economic Sciences and Information Technology,*
*PO Box 1174, Vanderbijlpark, 1900, Republic of South Africa*
*Email: malan.denheijer@nwu.ac.za/ roelien.goede@nwu.ac.za*

**Doi:10.5901/mjss.2014.v5n21p359**

***Abstract***

*The aim of this paper is to argue and demonstrate the use of a scriptable game-engine as a supportive technology for understanding and implementation in an active learning environment for computer science. It is argued that the use of an active learning environment as a suitable environment for computer science. As a case study, artificial intelligence is used as a representative subfield of computer science and implements an active learning environment with a scriptable game-engine for students. A demonstration is given of the use of a scriptable game-engine to teach intelligent agents using object-oriented programming concepts. The students were interviewed with general and specific open-ended questions in a written interview. Various Likert-type scale questions were also asked to rate their experience. Interpretive content analysis was used to understand their perceptions regarding the instructional design and their learning experience. Grounded in the students' and the authors' own learning experience, practical guidelines for the implementation of a scriptable game-engine as a supportive technology in any computer science classroom environment are provided.*

***Keywords:*** *computer science; intelligent agents; game-engine; object-orientated programming; artificial intelligence education.*

## 1. Introduction

Computer science forms part of many higher-education Bachelor of Science programmes. Computer science deals with the systematic study and implementation of algorithmic information processes (Denning et al., 1989). Many students find the abstract concepts of computer science difficult to master. There is widespread belief that supportive technologies that deliver improved visualisation, interaction and feedback will help students better understand the abstract concepts. Various visualisation solutions have been successfully used, but there is a concern that the time and effort to implement these may outweigh the benefits (Naps et al., 2002). The benefit will also only be gained if the technology forms an integral part of the student's active learning experience, and merely having it in the classroom will be of little use; hence the emphasis on supportive technology in a learning context and not merely technology (Bransford, Brown, & Cocking, 2000, p. 206).

This paper argues and demonstrates the use of a scriptable game-engine as a supportive technology for active learning in a computer science environment. To ensure the technological support forms an integral part of the active learning experience, the computer science discipline and the required characteristics of computer science graduates are discussed in Section 2.

The critical features of an active teaching environment with supportive technology are discussed, arguing its suitability for the computer science discipline (Section 3). Section 4 focusses on scriptable game-engine and what possible features can be used in a teaching environment. In Section 5 the focus is on scriptable game engines as technology for use in computer science teaching.

As a case study, the authors use artificial intelligence as a module with fourth-year IT students. Section 5 provides background knowledge on intelligent agents in artificial intelligence using object-orientated programming. Initially a learning environment was designed, which the authors believe will create an active learning experience, and they use a scriptable game-engine as a supportive tool for teaching intelligent agents (presented in Section 6).

The students were interviewed to establish if they had a successful learning experience and how the game-engine technology supported them in active learning. From the interpretive content analysis, feedback is given and guidelines provided that can be considered for any other computer science module and future research. Section 7 focusses on the

student perceptions.

Based on the experiences of the students and the authors own journey, practical guidelines are provided to implement supportive technology in the computer science classroom and improve the overall teaching environment (presented in Section 8). This section also includes the vision for future work.

Since the paper contains ideas that require a fair amount of technical details (which are considered to be outside the scope of this paper), two conference papers, one in press, on the technical details are referred to. In Den Heijer and Goede (2014a), it was demonstrated that an object-orientated scriptable game-engine can effectively implement a simple intelligent agent. In Den Heijer and Goede (2014b), it was argued that the use of the same method to teach object-orientation is conducive to teach artificial intelligence. The authors build upon this previous work and extend it to computer science.

## 2. Computer Science

The aim of the section is to position this work in the context of computer science. Denning et al. (1989) provided the classical definition of computer science, "Computer science and engineering is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation and application. The fundamental question underlying all of computing is, 'What can be (efficiently) automated?'" Algorithms are the most basic objects of concern and programming and hardware design the primary activities to implement these (Denning et al., 1989).

Denning et al. (1989) identify eight subfields or knowledge areas of computer science discipline as follow:

1. Algorithms and data structures
2. Programming languages and architecture
3. Numerical and symbolic computation
4. Operating systems
5. Software methodology and engineering
6. Databases and information retrieval
7. Artificial intelligence and robotics
8. Human-computer communication.

Algorithms are essential in all the other subfields of computer science (ACM & IEEE-CS, 2013). Thereafter programming is arguably the most important subfield of computer science and forms part of the implementation of most of the other subfields and serves as a bridge language to describe other concepts (ACM & IEEE-CS, 2013). According to Denning et al. (1989), each of the discipline subfields forms an underlying unity and each contains a substantial theoretical component, abstractions, design and implementation aspects.

Because of the rapidly evolving computing technology, the above list should not be considered as exhaustive subfields. Networking, the internet, safety and security has become part of modern computer science curricula (IEEE-CS & ACM, 2001). The discovery of natural information processes led to the change of the view that computing and information processes are purely man-made and that the that information processes might even be more fundamental than algorithms in computing (Denning, 2003).

Because of the rapidly changing fields and integration with other fields, students must be prepared for lifelong learning and not merely have factual knowledge (ACM & IEEE-CS, 2013; Denning et al., 1989). The broad, expected characteristics of computer science graduates include the following:

- Technical understanding of the different subfields
- Abstraction or modelling and the theory of complexity
- Understand the interplay of theory and practice
- System-level perspective of computer systems, from people, other domains to the individual components
- Problem solving skills
- Large project experience
- The ability of continuous learning throughout their future careers
- Appreciate the value of good engineering design principles.

It is evident that all of the above characteristics require deep technical understanding and the ability solve problems from various contexts. The most important characteristic is the preparation for lifelong active learning. The next

section discusses active learning environments and how they could support computer science teaching.

## 3. Active Learning Environments and Technological Support for Computer Science

The first discussion is on the requirements of a general teaching environment to create active student-driven learning, an understanding of concepts (and not the mere memorisation of facts) and transferable understanding. Supportive technology for the teaching environment, from a computer science perspective, is then examined.

### *3.1 Learning Environments*

The main goal of learning is to transfer that what was learned to other circumstances, either in the home, community or workplace (Bransford et al., 2000, p. 73). The critical features of a learning environment that affects students' ability to transfer knowledge are given by Bransford et al. (2000) and can classified into three main themes:

1. Understanding: Initial learning is essential to promote transfer (p. 53). Understanding the facts rather than memorising increases transfer (p. 55). Organising facts in a conceptual framework helps students to see patterns, relationships or discrepancies (pp. 16,17). Where possible, problems should be represented at a higher level of abstraction (pp. 63-65). Enough time should be spent on a single abstract concept with enough example problems before moving on to the next concept (pp. 20,58). Contrasting problems from multiple contexts must be provided as to let students understand when, where and why to use the new knowledge (pp. 60,62). Problems should ideally use real-world technological tools (pp. 74,77).
2. Feedback: Students and the lecturer must continually receive feedback on the level of their understanding (p. 59). The feedback can be instigated by the lecturer and drawn out by prompting and continuous formative assessment, or ideally by the students themselves using active learning. They should be encouraged to actively monitor their own understanding, evaluate strategies and consider resources (pp. 66,67). Active learning greatly enhances transfer and adaptive expertise enables lifelong learning (pp. 12,48,67).
3. Previous experiences: Because learning involves transfer from previous experiences (which is constructivism), students' pre-existing understanding and experience, formal knowledge or informal everyday knowledge should be drawn-out and built upon. Faulty understanding must be corrected, because it can hinder the understanding of new concepts (pp. 19,68-70).

The teaching environment must strike a balance between knowledge, formative assessments and the learner's previous knowledge and each area must complement each other (Bransford et al., 2000, p. 133). Focussing only on the learner's previous experiences will not necessarily help them to acquire the necessary skills. Focussing only on hands-on contextual assessments will not necessarily develop conceptual understanding and will limit transfer (Bransford et al., 2000, p. 22). The goals or outcomes must be aligned with what is taught, how it is taught and how it is assessed (Bransford et al., 2000, p. 151).

Understanding is the common core, and to reach a deep understanding, previous knowledge must be built upon, and continuous feedback must be implemented. Active learning enables students to take responsibility to monitor their own understandings and to be adaptive to new knowledge.

### *3.2 Supportive Technology*

Technology is required that can support the above critical features to achieve better transfer. Supportive technologies can be grouped in terms of their function:

1. Tools for understanding and implementation: These supportive technologies can be scaffolding tools that facilitate understanding or implementation tools for complex, real-world problems. Visualisation of information lets people more easily notice patterns and relationships of abstract facts, which leads to improved and deeper understanding (Bransford et al., 2000, p. 215; Naps et al., 2002). With interactive, visual simulations or models, students can implement more complex, real-world problems in multiple contexts.
2. Tools for feedback: Supportive technologies can be used to enhance two-way feedback in the classroom, provide opportunity to get feedback from experts and enhance formative assessments (Bransford et al., 2000, pp. 216-219). The internet can be used as a forum for students to give feedback to each other. The feedback tools will also make student's initial understanding visible, enabling new knowledge to be built upon initial knowledge. Feedback will also expose the level of difficulty students are experiencing.

The presence of these technologies must support the teaching environment, or else it can hinder the students and

be a waste of time and money (Bransford et al., 2000, p. 206).

### *3.3 Supportive Technology and Computer Science Teaching*

When the critical features of the learning environment, discussed at the start of this section, are compared with the characteristics required by computer science students (presented in Section 2), it is evident that all characteristics require deep technical understanding and the ability solve problems from various contexts. The most important characteristic is preparation for lifelong active learning. The critical features of the learning environment seem adequate to address all the broad requirement-characteristics of computer science students.

Especially with an abstract discipline such as computer science, students' previous knowledge must be built upon, or a suitable model must be given to start constructing their understanding upon (Ben-Ari, 1998). Depending on the curriculum, the previous formal knowledge or prerequisites of each subfield will determine the approach taken to address previous experiences of students. Programming, data structures and algorithms are prerequisites of many advanced computer science courses (ACM & IEEE-CS, 2013).

The inherent abstractness of computer science concepts lend themselves to interactive visualisation technology, but the students must be actively engaged in the computer science learning activity to gain any advantage of visualisation technology (Naps et al., 2002).

Because of the importance of algorithms and their implementation by programming, the requirements of a teaching programming language and the best practices for the visualisation of algorithms, respectively, will be considered.

A suitable teaching language and programming environment as supportive technology, should satisfy the following criteria (Kölling, 1999a, 1999b):

1. Clean concepts: Does the implementation in the language reflect the level of abstraction in the problem?
2. Object model and support: Is the model of execution simple and easy to understand? Does the graphical user interface (GUI) support instances of the code objects?
3. Easy transition: Would you be able to transfer the concepts used in language to another language?
4. Ease of use and the necessary integrated tools of the environment's Graphical User Interface (GUI).
5. Supports a high level of interaction, experimentation and visualisation of objects.

The conditions of clean concepts, the simple object model, visualisation and interaction satisfy the requirements of supportive tools for understanding (Bransford et al., 2000, p. 215). The conditions of easy transition, ease of use and the necessary integrated tools provide the requirements for supportive tools for real-world implementation (Bransford et al., 2000, pp. 207-209). Programming provides immediate feedback to the student, which is a major requirement for active learning (Bransford et al., 2000, pp. 12,19). Kölling (1999b) states that a good programming environment provides immediate feedback and reward to students and represents the most powerful support a teacher can wish for, "Every teaching environment should try to create this sense of fun through interactivity and immediacy."

Kölling and Rosenberg (2001) provide constructivist guidelines for teaching programmers to continuously build upon their knowledge: objects must be explained first, followed by filling in blanks in existing programs, adding new methods and classes and then finally developing a project from scratch. The importance of visualising the program structure and careful consideration of the user interface for interaction are also stressed (Kölling & Rosenberg, 2001).

Naps et al. (2002) provide an overview of the best practices specific for visualisation of algorithms:

1. Help students interpret the visualisation and its relation to the program elements.
2. Adapt to their knowledge level.
3. Provide multiple views, ideally a simultaneous visualisation and stepwise code view of the algorithm.
4. Provide performance information of the algorithms.
5. Support flexible execution controls such as play, stop and pause.
6. Support learner built visualisations to gain insight and a sense of responsibility.
7. Support for custom input data sets.

There exists a perception that the visualisation that might incur too much additional overhead (Naps et al., 2002).

From all the above discussion, it would be ideal if the students can use a real-world context tool for their problem-implementations and that the tool provides built-in interactive visualisation capabilities to enhance understanding. If it can provide feedback to the students, they themselves and the lecturer can gauge their level of understanding.

Scriptable game-engines, as supportive technology for computer science teaching are now presented.

## 4. Scriptable Game-Engine Technology

Some background on games and game-engines will first be discussed. The authors consider the definition of a game as type of play-activity in a rule-based, imitated setting where the player has to achieve certain goals (Adams, 2009, p. 3). To design and make a game, developers can reuse an existing game-engine or components thereof, which can include the rendering system, input system and collision detection system to name a few (Gregory, 2009, pp. 11,29). Custom game-objects with unique behaviours can be implemented if the game-engine has its own runtime scripting system (Gregory, 2009, p. 802). All of the systems of the game-engine are driven by the game loop event or tick (Gregory, 2009, p. 10). With each tick, every custom object's piece of code is executed and the rendering system draws the object to the screen (Gregory, 2009, p. 757). A game world editor is used to populate the game world with objects and set their initial properties (Gregory, 2009, p. 701).

The Unity game-engine uses a high-level object-orientated scripting language with class constructs to create fully custom game-objects (Goldstone, 2011, p. 40). Prior to the instructional design reported in this paper, the features of the Unity game-engine were compared informally with a mainstream Integrated Development Environment (IDE) from Java and C++. Regarding the teaching requirements and visualisation requirements from Section 3, the scriptable game-engine was by far superior, in the opinion of the authors. The initial main advantage was being able to reuse the game-editor's existing GUI for visualisation and interaction with the game-objects. The building of a GUI can be very distracting from the important implementation of the main program structure (Kölling & Rosenberg, 2001). In that case the tool will rather be a distraction and hindrance than offer support (Bransford et al., 2000, p. 206). The authors believe the similarities between Unity's C# object model and Visual Studio IDE will allow for better transition. This corresponds to the advantage of using real-world tools (Bransford et al., 2000, pp. 207-209).

## 5. Artificial Intelligence: Intelligent Agents

The demonstration of the use of a scriptable game engine as supportive technology in computer science is in the field of artificial intelligence and specifically intelligent agents. This section first provides a discussion of intelligent agents before focussing on an object-oriented implementation thereof.

### *5.1 Intelligent Agents*

The focus of this study is on intelligent agents, which look at AI from the viewpoint that intelligent behaviour is an end-result of rational decisions taken by an agent. An intelligent agent is a program that reacts to its changing environment while executing. The main goal of the agent is to solve problems (Luger, 2009, p. 266). It observes its environment and decides what the best action is to take (Russell & Norvig, 2014, p. 38). AI is a very suitable case module, because of its problem solving algorithms, the required discrete data structures and programming implementation. Linking back to the definition discussed earlier of computer science, problems are seen as an information process and an algorithmic process provides a solution (Denning, 2003). Artificial intelligence tries to solve problems (Luger, 2009, p. 2).

An intelligent agent has the following important features (Luger, 2009, pp. 266-267; Russell & Norvig, 2014, pp. 35-40):

- It observes the environment it is situated in and performs actions to achieve its goals.
- It performs autonomous actions without outside intervention.
- It perceives its environment and reacts to it. Its flexible and goal-directed using search, planning methods or pro-active learning.
- It can communicate with other agents to achieve the goal.
- The environment's properties include observability, randomness and how dynamic it is and has a large influence on how the agent implements its action.

An agent program implements the mapping of the agent's observations to actions on the environment (Russell & Norvig, 2014, p. 47). The agent program can use AI sub-techniques (tools) or any algorithm it needs to aid it (Russell & Norvig, 2014, pp. 58-59):

- Systematic search, either informed (heuristic) or uninformed: States of the environment are expanded and checked if they are the goal state. It's systematic, because it returns a complete path to the goal state before any action is performed. A* search is the most popular heuristic search method. These graph-search methods within the agent program requires a node tree discrete data structure (Russell & Norvig, 2014, p. 80).

- Logic, planning and learning: With insufficient agent knowledge, reasoning and learning modifies it to improve its performance.

Different agent designs vary only by how the agent program decides which actions the agent should take (Russell & Norvig, 2014, p. 47).

### *5.2 Object-orientated Implementation of Intelligent Agents*

In Den Heijer and Goede (2014a) it was shown that object-orientation can suitably implement a simple agent design in a scriptable game-engine. Object-orientation in software engineering has become the mainstream paradigm for most modern applications. It represents a program as various instantiated objects that send messages to each other. The objects' state and functions are encapsulated by the class construct. The class construct serves as a scaffold or blueprint to instantiate unique objects, each with their own state (Cohoon & Davidson, 2002).

There are criticisms against representing agents as objects alone, especially the requirement of the agents to be autonomous and being able to communicate with each other. Agent-orientated approaches provide a methodology to implement distributed multi-agent systems, but the problem is that agent-orientated techniques are not yet as mainstream as object-orientated techniques (Jennings, 2000). It is preferable to use a paradigm that is familiar to IT students' for implementation. Fortunately it remains practical to use an object-orientated programming language as implementation (Wooldridge & Jennings, 1998).

It was established in Den Heijer and Goede (2014a) that the main requirement for a game-engine to implement an intelligent agent will be a scriptable game-engine whose object model support full object-orientation, event-driven activation and coroutines. The authors chose the Unity game-engine because it uses mainstream C# or Java as a scripting language to customise objects' behaviour. The game-world editor possess all the required tools to design custom objects (Goldstone, 2011, pp. 17,28). The initial state of custom script-defined objects can be changed in the game-world editor's Inspector window, providing a way to interact with the game-objects (Goldstone, 2011, p. 113).

The implementation in the scriptable game-engine will be as follow (Den Heijer & Goede, 2014a):

- A single agent class script object is used to encapsulate the agent as a single, coherent problem solver.
- The agent program member in the agent object encapsulates the agent's design and behaviour, which invokes all the needed action methods such as searching.
- The agent program is invoked with each tick from the main game loop.
- The environment is represented as a separate class object in which the agent object is situated.

## 6. Instructional Design: Game-Engines for Teaching Intelligent Agents

The artificial intelligence module is optional for fourth-year honours IT graduates. The outcomes of the module state that students must be able to select, implement and analyse a suitable agent-design. The aim of this section is to implement an ideal active learning environment from Section 3 with a scriptable game engine as supportive technology in the computer science subfield of artificial intelligence.

### *6.1 Focus on Object-oriented Programming*

The IT students had a module that covered object-orientated programming in their first and second years. Starting with building on previous experiences, the students were given a refresher on object-orientation, which includes encapsulation, abstraction, the class construct and instantiation of objects. This was not only necessary because of the programming implementation of their problems, but because of the need to provide a conceptual framework to organise all the subsequent topics that will be discussed. Object-orientated programming is used as a bridge language to describe intelligent agents as objects, and the authors provide the required initial abstract learning before implementing any contextualised, real-world problems by explaining the agent object situated in the environment object. Den Heijer and Goede (2014b) provide a detailed discussion of how the technical components of agents are implemented in object-oriented concepts.

Using such an object-oriented framework, any new AI subtopic can be explained simply in context of an agent requiring another method or tool to solve a more difficult problem. These abstract concepts are all explained though direct classroom interaction before giving any assignments.

### *6.2 Teach Functionality of the Game-engine*

The game-engine's GUI needed to be explained to the students, and they were provided with resources on how to use the tool. It was important to ensure that the students did not view the game engine as a complication in their understanding of intelligent agents.

- The game-editor and scripting editor is explained (Figure 1).
- The object-model is explained.
- To help visualise the program structure, the students create a non-functioning agent class and environment class, each with a visual representation (Figure 1).
- The GUI's input/output is explained, especially the object Inspector that is used to change an object's initial properties (Figure 1).

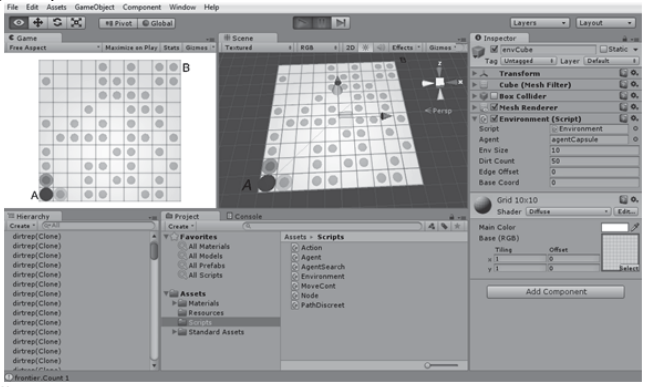


**Figure 1.** Unity editor GUI

### *6.3 A Project with Milestones*

The teaching strategy for teaching intelligent agents is built around a large project that is divided it into credit-bearing weeklong milestones (Den Heijer & Goede, 2014b). This is expanded upon as follows:

- The object-orientated implementation allows for complete classes or methods to be incorporated easily into a milestone.
- The current milestone method can provide the input parameters for the subsequent milestone method, as with the implementation of the search method. It allows for a progressive formulation, which is easier for students to understand.
- More time can be allocated to complete the milestones of more challenging topics.
- New abstract topics can be discussed before a corresponding a milestone assignment.
- Milestones are in context with the large project and not disjoint separate assignments; the students always know why they are implementing the respective class or method.
- Because milestones are credit-bearing, they force students who procrastinate to complete their work and creates a sense of responsibility.
- The complete solution to a milestone are given to the students after it is evaluated, letting them built upon a complete standardised implementation, even if they could not complete it. Abstract concepts are explained again if necessary.
- It allows for continuous feedback and regular formative assessment, which makes the students' thinking visible to them and the lecturer.
- It allows more time for the lecturer to prepare and more flexibility to focus on difficult topics.
- It follows the software development cycle, with design, implementation, and testing all according to a schedule with intermittent deadlines.

### *6.4 The project: A Vacuum Cleaner Agent*

For their project, the students must implement a vacuum cleaner problem. The vacuum agent is situated in a square 10m

grid environment with 50 random dirty tiles (Figure 2). They were only given vague guidelines and encouraged to consult online forums, code-repositories, manuals and the lecturers. After each complete milestone, it was demonstrated and given to them to build upon. The project is divided into milestones as follow (extended from Den Heijer and Goede (2014b) ):

1. Implement the environment, including the dirty tiles and visualise it.
2. Implement a reactive vacuum agent and let it find the closest dirt tile to clean it.
3. Compile a report on the results of the vacuum's time to clean the 50 tiles. Present the results and the visualisation to the class.
4. With the agent and environment objects functioning, upgrade the agent's abilities using a search method. Systematic search methods are explained. The agent must navigate through the dirty environment from one corner to the other (Figure 2).
5. Implement a node abstract data-type and a node-tree structure that any systematic search method can use. The A* graph-search method must be implemented by using the existing node-tree structure. Visualise the node-tree being expanded.
6. Compare the difference in performance of other search methods using the same visualisation and node-tree structure. Compile a report and present the results.

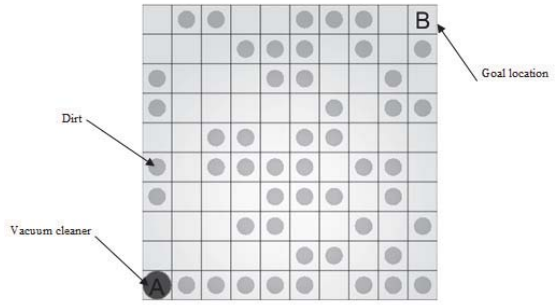


**Figure 2.** Initial 50 randomly generated dirty tiles

After the assignment and the completion of the intelligent agent part of the AI module, the students were interviewed.

## 7. Results from Interviews and Discussion

This work is part of an on-going study to improve the use of a game-engines in the AI classes by the authors. After the scholarly discussion, some of the initial findings of the work was published in Den Heijer and Goede (2014b).

This instructional design was implemented for the first time in 2014. This was the first time this module in artificial intelligence was presented in an existing programme to fourth-year IT students. Because it is a new elective module, only three students out of the thirty enrolled for the programme selected this module. The focus of this section is on the understanding of the experiences and perceptions of these three students. Written interviews were used, comprising both open-ended (qualitative) and Likert-scale type (quantitative) questions. Data analysis of the open-ended questions was done using interpretive content analysis. The results provided confirmation and valuable feedback for the instructional design of this module in future. The interviews focussed on the following areas corresponding to the discussion above:

1. Previous experiences and object-orientated conceptual framework for understanding: Building upon object-orientation to represent an agent-environment and its tools as discussed in Section 5.
2. The implementation of a contextual problem and building further upon it: The implementation of the reactive vacuum cleaner problem and its subsequent upgrading with search methods as discussed in Section 6.
3. The suitability of the Unity game-engine as a supportive technology: A tool for understanding (interactive visualisation) and a real-world implementation tool as discussed in Section 6.
4. The feedback provided by the milestone approach and the students' active learning strategies.
5. Transfer to real-world implementations, aspects the student find most valuable and skills they think they will use later on.
6. Overall, the students were asked explicitly if the game-engine either helped or hindered them.

The six fields are now discussed individually.

### *7.1 Previous experiences and object-orientated conceptual framework for understanding*

As explained in Section 5, the current programming skills of the students is a key factor in the motivation for using object-orientated programming for the implementation of intelligent agents. All three students that were interviewed reported to have had previous C# object-oriented programming experience.

The students were asked to explain intelligent agents as objects. They were asked open-ended questions to assess their knowledge on the theory of intelligent agents. The students demonstrated a clear understanding. One wrote, "Because the intelligent agent is an object, it can therefore be programmed accordingly. The agent needs an environment, which also is an object, and both of these can have states and behaviour." Another wrote: "In OO programming the paradigm is to create objects (these have data attributes, or metadata). …. OO programing also uses methods. Agent can do things (methods)."

It was concluded that the students perceive object-orientation as a suitable means of implementation. The students also demonstrated a clear understanding of the implementation of intelligent agents using object-orientated programming techniques. They have successfully employed their knowledge of object-orientation to enable them to understand basic agent theory.

### *7.2 The implementation of a contextual problem and building further upon it*

They were asked if the agent class (with its embedded tools) and its environment class (with its drawing methods) were a suitable representation of the vacuum problem and all strongly agreed. One replied, "Yes, the agent class and environment class in Unity helped me understand the difference between the two. The visual representation in Unity also helps one understanding how the agent and environment works." They were asked if the embedded search method within the agent was a suitable representation, another replied: "The abstract classes were used to display the expanding tree and was great at representing all its data types…. An agent is supposed to search for the path inside its head so the embedded search function is how it should work." When asked how much the assignment helped their understanding, one replied: "I did not really understand agents until I completed the assignment (the assignment did a good job in educating me)". The initial abstract explanation to them might have been unsatisfactory, but all the students replied that they did have a much deeper understanding of agents after they completed the assignment.

### *7.3 The suitability of the Unity game-engine as a supportive technology*

In order to understand how they experienced Unity's C# scripting as a teaching language and Unity's GUI as a teaching environment, the students were questioned according to the criteria of Kölling (1999a) Part 1 and Part 2. Their overall responses were favourable except for the steep learning curve of the game-engine (Table 1 and 2).

**Table 1**. Results for the suitability of Unity as teaching language and environment

| **Criteria** | Question | Sample response | Result |
|---|---|---|---|
| Clean concepts | Game-objects (GO) a good representation and motivate. | "Yes, the vacuum cleaner is a GO and the script attached to it (Agent class) gives you the ability to program what the agent should do." | Yes, clear understanding |
| Object model and support | Explain linkage between GO and class scripts. | "The linkage between the two is so that the programming in the class script which is of type Monodevelop can communicate and tell the game object what to do and how to behave." | Yes, clear understanding |
| Easy transition | Implement agents in other object-orientated languages? | "Yes, it's almost a direct import into Visual Studio C#. All that needs changing is you need to draw everything to make it visual and it will be in 2D. Unity looks way better than it would look in visual studio and it's much easier to make it look decent." | Yes; some effort; except graphics; prefer Unity |
| Ease of use and Integrated tools | Rate GUI, Editing, Debugging and Compilation of code, Interface learning curve, Integrated tools | GUI 4/5: "Because it is very user friendly and easy to see what is done." "There was no problem with Monodevelop and no other editor or tools were necessary. Monodevelop is easy to use and understand." " Working with Unity in a previous course allowed me to have a background with the Unity system." | GUI positive; Steep learning curve; Previous knowledge helped; Integrated tools |

**Table 2**. Results for the suitability of Unity as teaching language and environment

| Criteria | Question | Sample response | Result |
|---|---|---|---|
| Visualisation, interaction, experimentation | Game window helped in understanding. Node tree visualisation helped? Experimented and used inspector window? | "Yes, being able to visualise it inside the game-window one can physically see what is happening in the code behind."<br>"I wasn't really understanding A* search, but after doing it in Unity and visualising it, I understood it."<br>"It is difficult to see the fault in your code just by looking at it but looking at the actual tree growing you can see where your bug is." | Yes, visual; understand after visual; Visual debugging; Inspector- window positive |

Students were asked in which subfield of computer science the game-engine will enhance the learning experience and they respectably replied programming, databases and security.

From the above results, it could be concluded that the students perceive Unity as a suitable teaching language and environment. Because of the perceived steep learning curve, more time needs to be spent in the future on Unity or a separate Unity workshop might be beneficial before the start of the assignments. Also, in the view of the authors, some of them did not know how to systematically debug a program, on which some additional instructional time will be focussed.

The Unity game-editor seemingly provides a real-world implementation tool with its mainstream C# scripting and the added visual interactive front-end editor provides a tool for understanding. The visualisation additionally provided knowledge to debug the program.

### *7.4 The feedback provided by the milestone approach and the students' active learning strategies*

They were asked if they prefer the milestones to a receiving one large assignment and all three preferred the milestone method. One student named the distribution of effort as motivation for his preference of the milestone method. Receiving code after each milestone enabling them to be able to achieve the next milestone was the motivation for the other two students. In another question, the students were asked explicitly if object-orientation (separate agent and environment objects, separate search method) and the milestone divisions work well together. All students gave positive answers motivating the understanding that object-orientated programming implementation and the milestone method work well together.

They were asked how they knew their understanding was lacking, and all replied that through the practical assignment they became aware of it. They were asked what resources they used the most; all three replied Google, thereafter the Unity forums and classmates. All three replied that they very often used the resources.

The feedback provided by the regular milestones was found to be very useful. The assignments revealed where knowledge was lacking. The students made extensive use of the suggested resources and the authors conclude that they actively engaged in the learning experience. The authors believe the primary motivation for this was the fact that the milestones were credit bearing. Although the motivation was extrinsic, the students revealed that they did enjoy the challenge.

### *7.5 Transfer of Knowledge*

Table 1 presents results for among others, "Easy transition to other programming languages". Students were asked if they would be able to implement agents in other object-orientated languages and all three answered they would be able with some effort, but refer to the benefits of the visual representation and interactivity in Unity. They were asked what they thought was the most worthwhile skill they acquired after the assignment. Two of them answered programming and one of them replied the search methods.

### *7.6 Technology: Support or Hindrance*

As reported in the literature review, one should be careful to introduce technology that requires effort to use by the students. The authors were interested in the student's overall perspective on the matter. The three answers are:

"It definitely contributed to my learning of the subject. It was difficult to understand at the beginning of the semester but when I got the hang of it was far more interesting and understandable."

"At first it seemed very complicated to transform a concept e.g. the vacuum cleaner agent to a functional visual program. Having a little experience in Unity helped to get started. I think if I had no knowledge of Unity it would make it more complicated. After I have started the assignment and visualising it in Unity the concept become clear. At the end it

helped my understanding of the concept (more than it would have by just studying the theory)."

"It contributed a lot to learning how the work was done, visually seeing how one line of code could affect the work and seeing how it affected it made you understand why that line of code was there and how it contributed to the solution. If something did not work it was easy to see what went wrong and can easily find the code that was wrong and start testing it and help fix the problem."

The authors conclude that the students perceived the Unity scriptable game-engine as suitable supportive technology that contributed to their learning experience.

## 8. Guidelines, Conclusion and Future Work

The following broad guidelines are provided for those who wish to implement a scriptable game-engine in their computer science module.

An existing active learning environment that supports understanding and continuous feedback should first be put in place before even considering supportive technology (Section 3):

- Initial abstract knowledge should be transferred with students' previous experiences in mind. Ideally, a conceptual framework should employ the students' previous experiences.
- Real-world contextual problems should then be implemented to deepen and contextualise the initial abstract learning.
- Continuous feedback either instigated by the lecturer or ideally through the students will make their understanding or lack of it visible. A balance must be struck between promoting deep abstract understanding, feedback though real-world problem implementation and previous knowledge. Continuous feedback will prompt the student and lecturer to shift the balance to the area that requires most attention. The milestone system from Section 6 provides the necessary regular feedback and encourages active learning in the students. The milestone system functions well with the modular nature of programming implementations.

The authors are convinced that any subfield that must implement algorithms and data structures in a high-level object-orientated language such as algorithms, data-structures, networks and expert systems will gain an immense benefit from using a scriptable game-engine that employs a mainstream scripting language. The visualisation and interaction advantage will be gained with little additional overhead for the students or lecturer; only the additional methods to update the representation of the game-objects will need to be added. The game-engine will then function as a tool for implementation as well as a tool for understanding through the interactive visualisation. Subfields that require low-level, non-object based implementations or only require showing interrelationships will only gain the advantage of visualisation and interaction, but with additional overhead because the interaction will have to be scripted additionally. A tool should rather be chosen that best supports the structure of the subfield and careful consideration should be given to the additional overhead.

For a real-world implementation tool it should first be rated by the lecturer and then later by the students according to the guidelines set out in Section 3. Especially important is the gap between the real-world tool and supportive technology tool and the requirement for easy transition to the real-world tool.

Although there were only three students, this was seen as an opportunity to understand their experience of the module. Using their previous knowledge from object-orientated programming (Section 5), the theory of intelligent agents was cast to more familiar objects. The students were able to use the Unity scriptable game-engine that allowed them to do an object-orientated implementation that fitted into their prior understanding. From Section 7 they perceived Unity as suitably supportive technology. It was demonstrated that they had a successful learning experience from the instructional design and the Unity milestone assignments. They were very positive about the visual feedback and interactivity Unity provided them.

The authors are satisfied that they have argued and demonstrated that a scriptable game-engine can be used as a supportive technological tool in a computer science class, provided it forms part of the active learning environment. The perspectives of the students provided invaluable feedback and confirmation to continue using a scriptable game-engine in the module.

Future work will be pursued in the following areas:

- Implementing other AI topics and subfields in computer science using a scriptable game-engine
- Considering motivation and collaboration in the learning environment, and how game-engine technology can enhance it
- Implementing serious games as a supportive technology.

## References

ACM, & IEEE-CS. (2013). Computer science curricula 2013: curriculum guidelines for undergraduate degree programs in computer science: ACM. (Place of publication?)

Adams, E. (2009). Fundamentals of game design (2nd ed.). Berkeley, California: New Riders.

Ben-Ari, M. (1998). Constructivism in computer science education. Paper presented at the ACM SIGCSE Bulletin. (Incomplete - more information required, i.e. date and place)

Bransford, J. D., Brown, A. L., & Cocking, R. R. (2000). How people learn. Washington, DC: National Academy Press.

Cohoon, J. P., & Davidson, J. W. (2002). C++ program design: an introduction to programming and object-orientated design. New York, USA: McGraw-Hill.

Den Heijer, F. M., & Goede, R. (2014a). Implementing an intelligent agent in a known, observable, discrete and deterministic environment using a scriptable game-engine. Paper presented at the Intelligent Systems and Agents 2014 Conference (ISA 2014), in press, Lisbon, Portugal.

Den Heijer, F. M., & Goede, R. (2014b). Using a scriptable game-engine to teach intelligent agent artificial intelligence according to object-orientated teaching principles. Paper to be presented at the ISTE International Conference 2014, in press, Phalaborwa, South Africa.

Denning, P. J. (2003). Great principles of computing. Communications of the ACM, 46(11), 15-20.

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., & Young, P. R. (1989). Computing as a discipline. Communications of the ACM, 32(1), 9-23.

Goldstone, W. (2011). Unity 3. x game development essentials. Birmingham, UK: Packt Publishing Ltd.

Gregory, J. (2009). Game engine architecture. Boca Raton, Florida: A K Peters/CRC Press.

IEEE-CS, & ACM. (2001). Computing curricula 2001 computer science final report: the joint task force on computing curricula IEEE Computer Society Association for Computing Machinery. Place of publication

Jennings, N. R. (2000). On agent-based software engineering. Artificial Intelligence, 277-296. Volume missing

Kölling, M. (1999a). The problem of teaching object-oriented programming, Part 1: Languages. Journal of Object-oriented Programming, 11(8), 8-15.

Kölling, M. (1999b). The problem of teaching object-oriented programming, Part 2: Environments. Journal of Object-oriented Programming, 11(9), 6-12.

Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. Paper presented at the The Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001). Place and date

Luger, G. F. (2009). Artificial intelligence: structures and strategies for complex problem solving. (6th ed.). Boston, MA, USA: Pearson Education Limited.

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., & Rodger, S. (2002). Exploring the role of visualization and engagement in computer science education. ACM SIGCSE Bulletin, 35(2), 131-152.

Russell, S. J., & Norvig, P. (2014). Artificial intelligence: a modern approach (International Edition). Essex, England: Pearson Education Limited.

Wooldridge, M., & Jennings, N. R. (1998). Pitfalls of agent-oriented development. Paper presented at the AGENTS '98 Proceedings of the second international conference on Autonomous agents, New-York, USA. Date?