(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
28 August 2003 (28.08.2003)

PCT

(10) International Publication Number
WO 03/071383 A2

(51) International Patent Classification⁷: G06F

(21) International Application Number: PCT/US03/04161

(22) International Filing Date: 11 February 2003 (11.02.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/357,729 18 February 2002 (18.02.2002) US

(71) Applicant *(for all designated States except US)*: **INTERACTIVE SCIENCES, INC.** [US/US]; 22151 Little Pond Road, Kildeer, IL 60047 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **MARKLEY, Theodore, J.** [US/US]; 22151 Little Pond Road, Kildeer, IL 60047 (US). **GRUSON, Mathew** [US/US]; 5418 Cromer Place, Woodland Hills, CA 91367 (US).

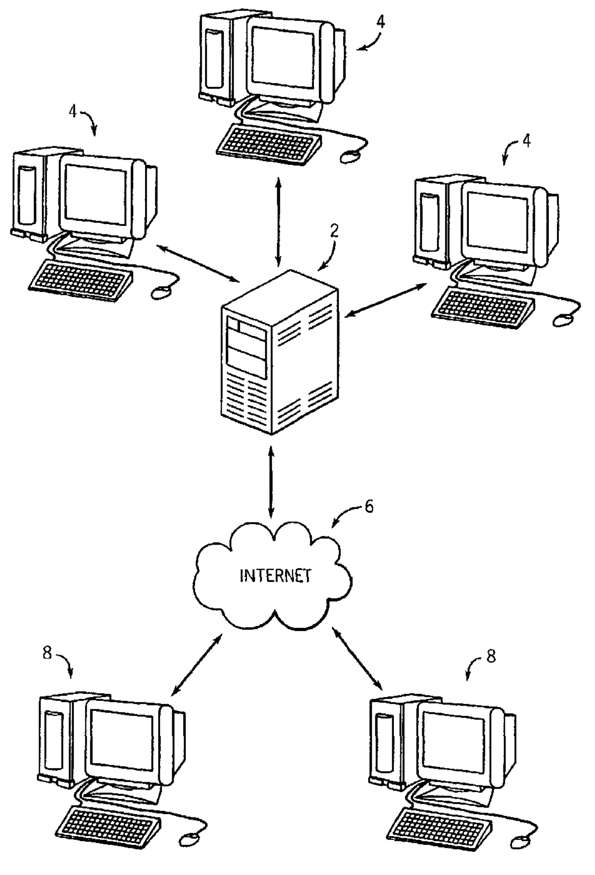
(74) Agent: **SAMMONS, Barry, E.**; Quarles & Brady LLP, 411 E. Wisconsin Avenue, Milwaukee, WI 53202 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE,

[Continued on next page]

(54) Title: DEVELOPMENT SYSTEM FOR MULTIMEDIA PROGRAMS AND PLATFORM FOR PERFORMING SUCH PROGRAMS



(57) Abstract: A game development system server stores a master asset pool which is available to a plurality of workstations. Each workstation includes game development tools tailored to the needs of game development team members and these are employed to develop game assets stored in the master asset pool. A game is built by selectively concatenating game assets and combining the resulting game program with a game platform program. The game platform program is installed on a user's general purpose personal computer to transform it into a dedicated game machine capable of running games from the game development system.

WO 03/071383 A2

ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

DEVELOPMENT SYSTEM FOR MULTIMEDIA COMPUTER PROGRAMS AND PLATFORM FOR PERFORMING SUCH PROGRAMS

BACKGROUND OF THE INVENTION

[0001] The field of the invention is the development of multimedia programs such as computer games and simulators. Such programs are characterized by their sophisticated graphic and sound presentation and a high level of user interaction through the use of a keyboard, mouse, joystick or the like.

[0002] Technology has done some wonderful things for the electronic entertainment industry, but these innovations have come with a cost. The escalating complexity of the programming involved in game production continues to create budget overruns, missed deadlines, and lengthening times to market. Programming becomes expensive as the software talent shortage continues. Game programmers command premium salaries well above their counterparts in other industries. Also, because of the complexity of today's products, often a significant amount of experimentation must be done during the programming phase to ensure that the product achieves peak performance, meets minimum quality standards, and works with the latest technologies. The reliability of newly developed technology is highly unpredictable, which places both development costs and schedules at risk.

[0003] On February 16, 2000 in Cambridge, Massachusetts, Massachusetts Institute of Technology held the nation's first conference to explore the role of video and computer games in popular culture. Conference organizer Professor Henry Jenkins declared that the event firmly established that "games are at an important threshold economically, technologically, culturally and aesthetically". At the event some "worried that the new processing power would push up the cost of game production and raise the technical expectations to the point that smaller companies or even mid-level developers would have trouble competing."

[0004] Developers creating electronic entertainment products are usually independent teams or companies whose primary desire is to create compelling, addictive games. Marketing, sales and distribution are all handled by a publisher. Many publishers have their own in-house development teams or divisions, called studios. Such studios usually consist of artists, programmers, and game designers. Sometimes, if they are large enough, the development team has in-house music and sound capability.

[0005] A new product begins with a game concept and proceeds through the design phase into production. Production consumes the bulk of the time and resources of the team. This is the period when the team falls behind schedule usually because of the complexity of software. The time to perform programming tasks is exceedingly difficult to forecast. Because the game industry is highly competitive, each new product must push the state-of-the-art as far as possible. This causes the programmers to venture into areas that are especially challenging. The single biggest reason for slipped schedules, missed released dates, and blown budgets is unanticipated technical surprises.

[0006] As the product nears completion, a quality assurance group scrutinizes the product looking for bugs. Quality assurance also assists the development team to balance the game play so that it is not too difficult (frustrating) or too easy (not fun). The length of time a product spends in quality assurance is in direct proportion to the level of complexity of the software and the technology base they use. If the software is derivative work (based on a prior released and well debugged product), the length of time can be shortened significantly. If the product uses a robust and well debugged technology base, this time can be reduced even more.

[0007] Games may be created for use on console game platforms such as those manufactured by Sony and Nintendo, or they may be created for use on personal computers using operating systems such as those sold by Microsoft Corporation. Because the console game platforms are computers dedicated to performing game programs specifically written for them, the user need only insert a CD into the console and begin playing.

[0008] The same is not true for computer games intended for use on a general-purpose personal computer. The game program must be loaded into the computer like any other program and this can lead to problems. These problems can occur more often with computer games because they make use of state-of-the-art graphic boards, sound boards, and other input and output devices that must be properly supported.

[0009] For the most part, game distribution is done using CDs, which are sold through the traditional retail channels. Most publishers would like to deliver over the Internet, but the bandwidth just is not there. Because console platforms are physical hardware, it is not possible to deliver game software to them over the Internet without significant redesign of the hardware, but an increasing number of personal computers have high speed Internet connections. Most computer games are 600 to 800 megabytes in size, with some reaching four times this amount. As a practical matter, such game programs are too large to distribute via the Internet with current technology.

SUMMARY OF THE INVENTION

[0010] The present invention is a game development system which aids game developers by decreasing the technical risk, decreasing their development costs, and decreasing their time to market. The invention also includes a user game platform, which creates a game environment on a general purpose personal computer that facilitates the playing of games produced by the development system by transforming it into a special purpose computer game playing machine.

[0011] The Development System provides game developers with a significant amount of the technology needed to develop computer/video games. This system provides a complete start-to-finish development environment. The development system consists of sets of tools, and a "development mode" version of the runtime platform.

[0012] There are different development systems (DS) for different game genre. All games have some common software elements such as mouse or joystick controls, menu systems, etc. In addition, specific game types or classifications, called "genres", have unique requirements that are not shared with other genres. For example, role-playing games (RPGs) calculate the results of a combat encounter based on rules that the game designer chooses. Virtually all role-playing games have these combat systems and therefore become unique software for that genre. Other software, such as a real-time 3D rendering engine, may be used in more than one, but not all genres. Each development system addresses the specific needs of a target genre, and is distributed to developers as complete "kits". Much of the code is shared between genres but this is transparent to developers. This allows one to reuse more code, and when a developer acquires subsequent a development system for a different genre, they will already be familiar with many of the tools.

[0013] The game platform is deployed on consumers' machines, and provides their personal computers with the capability of playing games developed with the development system. The platform is installed the first time they install any game developed with the development system and it is used to run subsequently installed games. The platform offers numerous advantages to the consumer: greatly simplified installation of subsequent games; consistent launcher interface; automatic online product updates; standardized product registration.

[0014] The development system includes sets of "role oriented tools" that assures each member of the development team has tools oriented to their role in the project. For example, a tool set for artists is written using artist-terminology in the interface (reduces training time), and do not require the artist to be a programmer. A tool set used by designers is written with their skillset in mind, and do not require designers to have artistic or programming skills.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] Fig. 1 is a pictorial representation of a game development system which employs the present invention connected through the Internet to personal computers which employ the present invention to play developed games;

[0016] Fig. 2 is a pictorial representation of a work station which forms part of the game development system in Fig. 1;

[0017] Fig. 3 is a block diagram of the work station of Fig. 2;

[0018] Fig. 4 is a block diagram of the game development system which resides on the server and workstations of Fig. 1;

[0019] Fig. 5 is a block diagram of a game platform and game which resides on a personal computer in Fig. 1;

[0020] Fig. 6 is a schematic representation of the layers in which a game program code is divided when produced on the game development system of Fig. 4

[0021] Fig. 7 is a pictorial representation of how game program code from different layers is combined to form game programs;

[0022] Fig. 8 is a pictorial representation of data structures in the game development system of Fig. 4; and

[0023] Fig. 9 is a pictorial representation of data structures in the game platform of Fig. 5.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

[0024] Referring particularly to Fig. 1, the preferred embodiment of a game development system includes a server 2 and one or more work stations 4. As will be described in more detail below, the server 2 stores a master asset pool and a number of programs that enable an operator at one of the workstations 4 to create a game program and produce it for distribution in a removable storage media such as a CD, or export it over the Internet 6 to a customer personal computer 8. Typically, however, a game program will be created by a team, and each team member will make their contribution through a separate workstation 4. For example, one workstation 4 may be configured for use by an artist who uses an artist tool set and programs such as that sold under the trademark "Photoshop" to create the scenes in the game program. Another workstation 4 may be configured to use a game designer tool set such as a game logic scripter program, and yet another workstation 4 may be configured with a programming tool set such as a C++ program editor.

[0025] Referring particularly to Fig. 2, the workstation 4 includes a mini-tower 10 which houses a processor and associated circuitry, memory, and peripheral interface circuits. One of the peripheral devices is a commercially available CRT monitor 12 which connects to a graphics circuit housed in the mini-tower 10, and another peripheral device is a keyboard and mouse 14 that connects to a PCI-based controller in the mini-tower 10. An operator may input data through the keyboard and control the position of a cursor on the monitor display 12 using the mouse. Typically, the workstation 4 also includes loudspeakers 16 that are driven by a PCI-based sound card in the mini-tower 10 and one or more special purpose input devices such as a joystick (not shown).

[0026] Referring particularly to Fig. 3, the workstation 4 includes a processor 20 which executes instructions stored in a memory 22. The processor 20 is a commercially available device such as that sold by Sun Microsystems, Inc. under the trademark UltraSPARC-IIi. It incorporates on-chip memory and I/O control to facilitate system integration. It is a superscalar processor implementing the SPARC-V9 64-bit RISC architecture and executing the instruction set sold commercially under the trademark "VIS". It also includes an integral PCI bus driver which provides a direct interface with a 32-bit PCI bus 24. It also includes integral memory management circuitry for handling all external memory 22. Other general-purpose computers can be used, such as computers based on Intel, Advanced Micro Devices and Motorola microprocessors.

[0027] The PCI bus 24 is an industry standard bus that transfers 32-bits of data between the processor 20 and a number of peripheral controller cards. These include a PCI EIDE controller 26 which provides a high-speed transfer of data to and from a CD ROM drive 28 and a disc drive 30. An Ethernet controller 32 supports data transfer with a number of peripheral devices, including input from the keyboard and mouse 14 and communication with Ethernet ports on the server 2. A graphics controller 34 couples the PCI bus 24 to the CRT monitor 12 through a standard VGA connection 36 and a sound card 23 couples the PCI bus 24 to the loudspeakers 16. Many other devices can be connected to the workstation 4 using an appropriate PCI compatible interface circuit and associated driver software,

[0028] Architecturally the personal computers 8 used by customers are very similar to the workstations 4 described above, but they typically employ microprocessors manufactured by Intel Corporation or Advanced Micro Devices Corporation, and employ a "windows" operating system sold by Microsoft Corporation. The particular hardware, such as the sound card and graphics controller card, used on a customer's personnel computer 8 may vary considerably in capability and features. One of the objects of the present invention is to insulate the game program from such variations by providing a game platform program resident on the customer's computer that will be described in more detail below.

[0029] Referring particularly to Fig. 4, the game development system includes an asset manager 50. Typical games today can contain tens of thousands of assets in the form of graphic images, sound files, code etc. An asset is a game related element that is unique to the game and gives the game it's personality—for example, a drawing that is the background for a particular scene is an 'art asset'; a sound that is heard in the game is an 'audio asset', etc. The asset manager 50 operates identically for all genres, and includes tools to manage the project, track revisions to code and data, and allow each development team member to test their own changes (code or assets) in the actual game without programmer assistance and without interfering with the work of others. It allows anyone developing assets to see how those assets perform in the actual product instantly. This instant feedback loop allows an artist to make sure their assets behave as expected without having to wait for an integration phase. This shortens the error "detect and correct" loop to the point where people developing assets can be assured that their components are working properly without the aid of a programmer. To accomplish this, the asset manager keeps local copies ('local' meaning on the particular user's workstation) of all assets in the game. When a developer creates an asset (or modifies an existing one) the new information is kept locally, and only affects the local version of the game. Thus, an artist can create a new background, and play the game with their local asset pool to see what it looks like. None of the other team members will see this new background until the team member who created it integrates it into the Master Asset Pool via the Update/Revision Controller (62). In this way, if a programmer were to be testing code that is not fully debugged, it will not interfere with the work of the other team members. Likewise, if an artist is creating a new animation and is testing it in the game (their local version of the game), the unfinished work will not interfere with anyone else's work.

[0030] Each game program developer has access to two asset pools. A master asset pool 52 lives on the server 2, and contains the master copies of each asset, and a local asset pool 54 lives on the developer's workstation hard drive 30, and is used to test changes to the game. The two asset pools are automatically synchronized with each other, removing any chance of a developer "forgetting" to update critical files they have worked on.

[0031] Industry-standard tools 56 with which developers are already familiar are fully supported. Assets created with these tools 56 are "imported" into the system when created, and re-imported if updated. Revisions are tracked allowing a developer to "rewind" code or assets to a prior state. Artists, game designers and sound artists think and work different than programmers and one aspect of the present invention is to tailor the developer tools to the particular "talent-types" on the development team. This tailoring is done by creating user interfaces that follow the thinking pattern of each different type of talent. In addition, the language and terms are tailored to each talent type. By using plug-ins to the tools that each talent-type uses, we keep the talent-type in their own world or domain of understanding. For example, the interface terminology is appropriate to the user: A programmer's tool might refer to a single piece of art as a "bitmap" whereas the artist's tool refers to it as a "cell".

[0032] For each class of asset in a project (bitmap, sound file, animation, etc.) an asset importer 58 handles the task of moving the asset into the local asset pool 54. The importers 58 use an asset database 60 to keep track of which assets have been updated in the local data pool. Where possible, these importers 58 are "plug-ins" to standard tools, allowing someone using (for example) PhotoShop to import a background image into the system without leaving PhotoShop. These importers 58 convert standard formats into common system formats. Metadata that is not part of the original file may also be included. Any preprocessing that will save time at runtime (i.e., speed up the game) is done by the importer 58. Where appropriate, data compression is also applied.

[0033] The asset database 60 contains only metadata (the actual assets are stored in the asset pools). The database 60 stores data which enables one to know exactly how many assets are in the game being developed, where they are, who (if anyone) is working on them and has them locked, who revised what when, and what language (if any) the asset was created for. Most of the tools access and/or update this database 60 and this information is used by the asset manager 50 to keep track of all project assets (which can number in the tens of thousands for highly-complex game projects.

[0034] An updater/revision controller 62 couples the local asset pool 54 on a game developer's workstation 4 with the master asset pool 52 on the server 2. The job of the updater/revision controller 62 is to synchronize the master asset pool 52 with the local asset pool 54 for each developer. It also synchronizes raw assets used by tools that are not included in the final build (i.e. script "source code", raw meshes and maps before rendering, etc.). This synchronization assures that each game program developer employs the latest assets from the master asset pool 52, and it enables a developer on one workstation 4 to "check out" and lock specific assets they are going to be working on.

[0035] The controller 62 also enables a developer to test assets they are working locally at a workstation 4 without affecting the master asset pool 52 and then "check in" and unlock an asset that has been updated and tested. The controller 62 also maintains multiple "rewind" files, which enables earlier versions of assets to be retrieved in case a mistake is made or an asset is accidentally ruined. The controller 52 also maintains a history file which indicates who has checked out and modified each asset and when.

[0036] The information maintained by the updater/revision controller 62 is used by a project manager program 64 which allows developers to set and check various project-wide attributes. The project manager 64 enables a developer to set which project they are working on. Every tool in the system knows which project a particular developer is working on at any given moment. If an artist (for example) is working on multiple projects, a couple of clicks is all it takes for them to switch projects and work on something different without mixing up assets between projects. It allows new projects to be created. It allows a project manager to see statistics on their asset pools 52 and 54 and it allows a project manager to define new language sets (i.e. add Brazilian Portuguese to the list of languages the game is being developed for).

[0037] Most of the tools used by game developers on the team are specially developed; or adapted tools for the particular talent-type. However, other general purpose tools, such as a C++ program editor, are also provided for more sophisticated developers who desire to write lower level, game specific routines. Microsoft C++ is the base compiler language supported by the system because of its universal acceptance as the compiler of choice for game developers. Where appropriate, the tools are integrated into Microsoft's Visual Studio, allowing developers quick access to development system functionality. The tools are also capable of operating in a standalone environment, so Visual Studio is not actually required, and developers who prefer other integrated development environments (Borland's C++ IDE for example) have access to all functionality. The runtime code is developed in MSC++ and assembly language (where appropriate). Developers have API level access to all the modules in the game platform software. An API, or Application

Programmer Interface, defines how to communicate with the software modules. It defines the inputs and outputs and how to manipulated them to achieve the desired function provided by the module. API access through C++ code gives more control, but requires a more thorough and technical understanding of the system, whereas the access to this functionality using a tool (e.g., accessed in a simple scripting language, or by manipulating a tool) is far easier, but more limited. For example, a developer can say "play the gunshot sound here" with a tool, without any programming. A programmer on the other and can use the API to "play the sound, on this channel, with this priority level, and when it's done playing, trigger an event called <whatever>". The latter is far more sophisticated, and only accessible via an API. Most developers won't need this level of access, and can do most of what they need through the tools.

[0038] All of a game's assets can be grouped by their logical location in the game being developed. For example, a game can be divided into scenes, pages, rooms, levels, etc. Because the grouping nomenclature changes from genre to genre, the universal term for a collection of assets grouped by their placement in the game is the "section". For example, a menu screen developed with our menu generation tool will have numerous assets associated with it (artwork, sounds, animations, etc.). Since those assets all appear in the same section, the runtime code that displays the menu can load them into computer RAM during runtime as a group. The developer does not need to load each individual button and sound effect.

[0039] This is accomplished by physically storing the assets in each section together in the master asset pool 52. The resulting section of assets offers a number of advantages. Games distributed online can be downloaded a section at a time (a section may be a "level" in a "twitch" game, or a "scene" or "act" in a story game). Games can span multiple CDs/DVDs and the developer will not need to worry about when to prompt for the insertion of a new disc. Games can be partially installed, having only those assets and modules that manipulate the assets which should run instantly (like opening screen), or those requiring hard-drive speeds (such as full-screen animations) installed on the hard drive. Download versions of games can be created that differ from their CD-delivered versions by eliminating entire sections of large-sized assets (i.e. the download version may not contain the full opening animation). The code is identical, essentially using a simple "if the opening animation asset file is present, play it, otherwise don't" algorithm.

[0040] When a game is to be assembled into an integral program, an asset concatenator 66 is employed to selectively build the game program. The asset concatenator builds the final distribution data files 68 (.HAGs) from the master asset pool 52. Different build scripts can be developed for different types of builds (i.e., demo vs. full game). Only assets needed by the desired build type are included in the final .HAGs. This program is also aware of which language is being built, and will include the appropriate assets for the target language. The final .HAG files contain all the information necessary for a particular version/type/language of the game program. Further data compression is performed at this step if appropriate. The HAG files are concatenated/compressed game assets. The HAG files are decompressed during game play and turned back into assets for use with game platform modules.

[0041] The operation of the asset concatenator 66 is scripted, and it can use multiple build scripts 70 to build different versions of the game from the asset pool 52. Building a demo version of a game involves only a couple of mouse clicks. Assets and code common to both the demo and the full game are not duplicated, removing the problems inherent in code divergence – if a bug is fixed or an improvement is made in a part of the actual game that appears in the demo, the demo is updated automatically.

[0042] All assets in the pool 52 are tagged by their language, or by their non-language-specific status. Games are sold worldwide and when for example, a game is used in Germany any assets in the game that use English must be converted to German. By tagging which assets are language dependent we can quickly identify those assets so they can be changed when a language conversion is needed. In addition, often a game will ship with more than one language. For many of the assets there must be a version for each language the game supports. At the start of game-play the consumer selects which language they want to use. The assets must have a way to identify them so the proper one can be loaded during game play. The assets are kept in parallel asset pools. The different language-specific-assets all have the same filenames so when a language option program 72 creates a version of the product in (say) French, no code has to change. Other programs check the asset pool 52 and report on which assets need to be localized, and report on the completeness of any particular language. Once the localized assets are in the system, building a version of the game in a different language is no more complex than checking the "French" box and hitting the [Build] button.

[0043] The final game product is produced by a project builder program 74. It produces the final master for storage on a CD or DVD, or for downloading through the Internet. It is also script driven and is responsive to the language options 72 and the type of game (e.g. demo/full game) being built. The project builder 74 assembles the .HAG files 68, the installer program 76 and the game platform program and data files 78. It also assembles ancillary files 80 containing such things as readme.doc files and autoplay files.

[0044] When the developer creates a final "build" of the game, the project builder 74 encrypts the resulting files. These files are the actual game assets, and are the data read by and manipulated by the game platform modules in the customer's personal computer in order to play the game. These files are encrypted with a "then current" private encryption key specific to the particular game program developer. The versions of the runtime platform modules that are included with that build as part of the platform files will hold the decryption key, and security servers will record which keys were granted for which game build. At runtime (when the end-user plays the game) the files must be decrypted in order for the game to work.

[0045] One of the popular ways of wasting time in game program development is making mistakes when creating the "final build". Developers normally manually compile assets, move them to a staging area, then add all of the various files needed for the build. With a manual process mistakes are quire common, (especially for an overworked developer on a deadline). The above-described automated build tools allow the project manager or lead developer to produce a burner-ready product image with a single mouse click.

[0046] During both the development phase and the on-going maintenance phase of a game product, the runtime code can be executed in a number of different modes using a test program 82. The test program 82 can be configured by the developer to assemble runtime assets from a local pool 54, the master pool 52, or from .HAG files 68. This allows a developer at a workstation 4 to run the game locally, using their local asset pool, using exactly the same platform modules and any game script assets as the final game and platform. This allows far faster and easier debugging, and eliminates hard-to-fix bugs that only occur in the final build but are not reproducible in test modes.

[0047] This test program 82 also provides developers with the ability to easily meet the need to frequently produce "in-house demos" (i.e., showing project manager, executives, investors, etc. their progress). A snapshot can be taken of the current platform modules and asset pool, and copied to another folder or a CD, which will operate entirely on its own without requiring a final installer or even a final build.

[0048] To better understand the structure and operation of the game program that is developed using the system depicted in Fig. 4 and run on the system of Fig. 5, the "stratified" runtime program code must be understood. Referring particularly to Fig. 6, the runtime program code needed to perform a game on a personal computer 8 is stratified into four layers 100, 102, 104 and 106. As will become apparent from the discussion below, some of these layers of code form genre specific "engines" which reside in the game platform resident on the customer's personal computer 8 and other layers of runtime code are specific to a particular game and must be distributed in some manner to the customer's personal computer.

[0049] The lowest level of runtime code is in layer 1. Layer 1 code includes low-level drivers designed to take advantage of the benefits of a particular hardware/operating system platform. There is one implementation of this layer written for each such supported platform. The interface to all low-level routines is defined such that an implementation of this "hardware abstraction layer" for the X-Box requires minimal changes to the upper layers. In this way, porting a game from the PC to the X-Box (for example) requires less time than porting other games.

[0050] These are the lowest-level routines, and generally consist of API wrappers for machine-specific functions. Regardless of the platform, the APIs for modules in layer 1 will be nearly identical as seen by layer 2. Example layer 1 modules include the following.

- 2-D display manager – Controls 2-D displays (including output from 3-D routines). Support retrace synchronization, multiple layers, and is tweaked for the capabilities of each platform.
- 3-D display manager – Handles the lowest lever of 3-D image generation. This layer will be extensive, but will also be tweaked for the capabilities of each platform. Note that in this context, the PC will actually be considered multiple platforms depending on which third party 3-D hardware is in use.

- Sound manager – Controls audio output. Support multiple channels, surround sound, 3-D audio imaging, sound layering and mixing and other capabilities of the supported hardware.
- Input manager – Handles all input devices (mouse, keyboard, joystick, flight yoke, etc.).
- Memory management – Used primarily by the Asset Manager (in Level 2), performs the low-level allocation and deallocation of system memory. Platform-specific quirks (garbage collection, heap defragmentation, etc.) are handled automatically.
- File I/O manager – used primarily for non-asset related read and write operations. For example, the "save game" function of a PC will involve file writes, while the same thing on a console will involve read/write to/from EEPROM and/or memory cards.
- COM I/O manager – for client/client and client/server communications. These low level routines will involve the establishment of connections, transfer to information packets, and will handle the platform-specific aspects of inter-machine communications (i.e., support for a PC on a LAN, an X-Box modem, etc.).

[0051] The level 2 layer runtime code contains universal routines used by all games in all game genres. This code is used regardless of the hardware/operating system platform. Because it is universal, there is only one implementation of this runtime code and most of the engine's runtime code will be written in this layer. This layer, along with layer 3, contain the APIs developers are advised to use. Some of the items in this layer appear similar to those in layer 1; the difference is that these layer 2 items are non-platform-specific, and perform higher-level functions. These routines are further divided into sublayers, allowing developers to choose how intimately they use each function. If, for example, one of the layer 3 routines provides 95% of the functionality a developer needs for a particular operation, they can achieve the other 5% by accessing the next deepest layer. Note that the 3-D support at this level is for basic object rendering only. First person "run through the dungeon" 3-D considered genre-specific (first person shooters), and is therefore part of layer 3.

[0052] Low-Level Layer 2 Modules include the following.

- 2-D Display manager – handles the high-level aspects of placing images on the screen. This layer is where the matte management is performed (resolving layers, updating only portions of the screen that have changed, etc.).

- Simple 3-D manager – Allows rendering of simple 3-D objects. These functions are primarily intended for 3-D support in primarily 2-D environments. For example, rendering houses and buildings in a sim, or rendering an animated 3-D object in a 2-D scene will be accomplished by these functions.
- Audio manager – These non-platform-specific routines allow the playing of music and sound effects. Layering, mixing and 3-D imaging are supported.
- Input manager – Gives higher-level routines (and the developer if desired) access to all input devices. As an example of how the code is layered, at this level there exist events such as "key pressed" or "mouse moved left". In layer 3, the developer does not need to worry about which keys/devices are assigned to what actions, and deals with events like "player moved left" or "player fired" (no need to know what key/button that happened to be).
- File I/O manager – handles the retrieval of data from the file system. Normally the developer will not use these functions since the asset manager handles most loading operations, however if they wish to manually open a file, the I/O manager will make sure all of our development modes are supported (i.e., its transparent to the developer where the data are coming from: local asset pool, master pool, etc.).
- Asset Loader – These functions allow a developer to manually load (or unload) an asset. Normally asset loading is done by the asset manager, but it a developer wants to manually load an asset (like config or .INI file they developed) this functions provided the ability.
- COM I/O manager – the low-level layer 2 version of the COM manager allows the developer the ability to manually open and close connections, and to manually transfer data packets. Normally they would use higher-level functionality.
- Memory management – The low-level 2 memory manager allows the developer to allocate/deal locate memory for their own purposes. They may be generating data tables not supported by the development system; this function allows them to allocate memory for their own use.

[0053] High-Level Layer 2 Modules include the following.

- Asset manager – the high-level layer 2 asset manager allows the loading and unloading of individual assets, or of all assets required by a particular section of the game (a "section" could be a screen, page, scene, menu, game level, etc. depending on the genre of game being developed).

- Runtime support for tool-generated objects – this group of routines allows the developer access to the game objects they have created in tools. For example, if they have built a complex menu screen suing a tool, they can call up that menu from here. Advanced developers can also derive custom objects for the tool-generated objects, giving them a great deal of power and flexibility.
- Animation objects – the highest level of access to the animation system. Animations that are not part of a tool-generated object can be loaded, run, stopped, unloaded, etc. The developer can use existing animation objects, or derive their own from out base classes.
- Load/Save operations – Provides high-level access to "save game" and "restore game" functionality.
- Access to platform functionality – The API allows the developer to get information from the modules in the platform itself. The platform supports such functions as "has the user registered this game?". These are platform functions since the platform handles product registration. It may be helpful to know if the user is registered, their name, their preference settings, etc.

[0054] The level 3 layer runtime code contains functions specific to a particular genre of game. For example, character management is important to a role playing game, but not to a simulator game. There is one implementation of layer 3 runtime code for each genre of game supported. Examples of code found in this layer for a role playing game are:

Map manager;
character manager; and
perspective display engine.

[0055] Examples of code found in this layer for a 3-D shooter game are:

3-D display engine;
enemy AI modules; and
projectile/character collision detection.

[0056] The level 4 layer runtime code includes routines written for a specific game. Code is placed in this layer when there is no plan to reuse it in other games. An object of the present invention is to keep the runtime code in this level 4 layer to a minimum. That is, when a game (layer 2) of a particular genre (layer 3) for a supported hardware/operating system platform (layer 1) is to be created, only the runtime code is needed to distinguish the game in an artistic and game rules sense from other games in this genre. This ability to reuse significant amounts of runtime code is illustrated in Fig. 7.

[0057] Not only does this layering reduce the burden as a whole on the game development team, but it places more of the burden on the creative artists and game designers rather than the highly technical programmer team members. This also means that if a customer has purchased a previous game of the same genre, only the newly created layer 4 code need be installed in his or her personal computer 8 in order to run the new game. This makes the installation of the new game a trivial undertaking, and it makes practical the downloading of the new game to the customer over the Internet because of the reduced file size.

[0058] To take maximum advantage of these benefits of runtime code layering a set of programs referred to herein as the game platform is installed in the customer's personal computer 8. As indicated above with reference to Fig. 4, the game platform files 78 are included with each produced game and are installed on the customer's computer 8 when the game is first installed. This initial installation of the game platform is similar to any program installation on a personal computer in which hardware and software conflicts must be resolved. However, once the game platform is installed with properly operating layer 1 and 2 runtime code, subsequent game playing with different layer 3 and/or layer 4 runtime code is a simple procedure. Once installed, the game platform transforms a general purpose, personal computer 8 into a single purpose game machine much like those manufactured by Nintendo and Sony. The user simply "clicks" on the game platform desktop icon and the program opens by presenting a welcoming screen with a menu of options. These options include the games to load which can be played by merely selecting it from the menu, and if needed, inserting the game CD/DVD in the CD/DVD drive 28.

[0059] Referring particularly to Fig. 5, the game platform maintains a local module database 200 on the customer's personal computer 8 which includes all of the runtime code loaded from previous games as well as new or updated runtime code and associated data structures from the current game. This local data base 200 is maintained by a module update manager program 202. To better understand the game platform operation, the operation of the module update manager program 202 will be explained first in detail.

[0060] The module update manager 202 assures that the customer has the latest versions of all the runtime code modules needed for all the games they have installed. It performs several functions:

- updates general system components (including itself);
- updates game-specific code and data ("patches");
- replaces entire .HAG files, or for speed, downloads only new assets and rebuild the .HAG files locally (saves a great deal of time for the user);
- performs CRC checks of any security modules, and replaces them if necessary.

[0061] The game platform supports multiple copies of runtime code modules designed for different games. This makes it possible to download an updated module that includes updates that are intended for specific games, without changing the modules used by games that do not require the update. This relieves the game developer of the responsibility of making sure every update module works with every game that has already shipped.

[0062] The module update manager 202 does not care where the updates are coming from. They will usually come from the Internet, but space permitting, each game CD/DVD product includes the full set of updates available at the time of production. The module update manager 202 can read an updated module used by a previously installed game #1 from a CD just inserted to play a new game #2.

[0063] The module update manager 202 relies on a master database stored on the server 2. Referring to Fig. 8, this database contains information 204 on each game and information 206 on each module. The local database 200 kept on the customer's personal computer 8, contains only information pertinent to the games they have installed. The master database is accessed via the Internet whenever the module update manager 202 in a game platform wants to check for updates. There is an entry in the game product table 204 for each game ever developed for the game platform. The available module table 206 contains information on every runtime code module ever developed for the game platform. Each product refers to all the modules which it uses, and for each product, the versions of each module that have been certified for it are known. This allows the game platform to understand which games need which modules, and more importantly, which modules may NOT work with certain games. Sometimes a new 'module' will not be compatible with an older game. One key feature of our system is that a game will only use those modules for which it is certified, and will not attempt to use modules for which it is not certified, EVEN IF the other modules are newer.

[0064] Referring particularly to Fig. 9, the local module database 200 on the customer's computer 8 contains information 208 on each game product they have installed. A required module table 210 keeps track of which modules are required by each game product; there may be multiple records for the same module because each game may require different versions of that module. An installed module table 212 keeps track of which modules have been installed on the customer's machine; there is only one entry for each module installed.

[0065] By way of example, the product table 208 may contain an entry for Hollywood Tycoon. That entry points to a list of required modules in the required module table 210. It may point to the joystick controller module record, which has recorded that Hollywood Tycoon can use version 4 or version 5 of the joystick controller module. The installed module table 212 indicates us that version 5 of the joystick controller module is installed, so that particular module does not need to be updated. On the other hand, if Hollywood Tycoon needs version 6 and that version is not listed in the installed module table 212, the module update manager 202 knows an update is required.

[0066] Once the module update manager 202 determines that an update is required, the update process is performed. This can be done using the Internet, or it can be done using the CD or DVD media on which the current game is stored. A list of modules to be updated is created and the customer is notified that updates are available over the Internet and told approximately how long it will take to download them. An "advanced user" option allows computer savvy users to select which individual updates they want to install, but in most cases all available updates will be installed. The updates are downloaded, added to the installed module table 212 and installed. The tables are scanned again, and any modules which are no longer required (i.e. those that have been outdated and are not required by any older games) are removed.

[0067] Each product/game distributed for this platform includes on the CD/DVD media as many module updates as space will allow, as well as a then-current copy of the available module table 206. The same process described above takes place, only the updated modules are taken from the local removable media (CD, DVD, etc.). A date-check assures that even if someone installs a very old game, containing an outdated available module table 206 and outdated modules, they will never overwrite newer modules with older ones.

[0068] One of the unique features of this system is that is allows multiple copies of the same module to co-exist on the game platform. If an older game works just fine with an older module, it can continue to do so, even if a newer module is installed. Unlike most operating systems and driver update processes, the memory update manager 202 will never render an older game inoperative. The only way an older game will use a newer module is if that game has been certified to use that module as reflected in the required module table 210.

[0069] An “update” can take several forms. Usually, it is a complete modules update, consisting of single files, which are downloaded and used as-is by the game platform. These updated modules may include part of the game platform itself.

[0070] In some cases, a small part of a large file will need to be updated. As discussed earlier, game code and data is kept in large concatenated files (.HAG files). A game may be updated by a developer such that new runtime code and new data in the .HAG file are needed. The game code is updated in the usual fashion, but if only 1 megabyte of a 50 megabyte file needs to be updated, there is no need to download a full copy of the new file (downloading 50 megabytes is time-consuming). Instead, “patches” can be downloaded. The downloaded “patch file” is itself a small .HAG file, containing new and/or updated code and/or data. The module update manager 202 recognizes the patch file as such, and rebuilds the main .HAG file in the game platform using the new information from the patch file. In this way, a huge file can be updated without having to download it in its entirety.

[0071] There are times when multiple updates are required as a group. For example, if game code is updated, and the new code uses new assets that appear in a patch file, the user must install neither or both updates in the game platform in order for the game to operate (if they install only the new code, it will look for assets that are not present and will not work properly). Such updates are considered "entwined". Entwined updates must be downloaded together, and the module update manager 202 user interface (the "advanced user" screen) makes this clear, and makes it impossible for the user to select less than all of the elements of an entwined update. For the non-advanced user, such updates are completely transparent, and all they know is that the updates work properly.

[0072] The update process is designed to be tolerant of Internet connection problems. Partially downloaded updates will not be installed. Should the user lose Internet connectivity in the middle of an update (or if they accidentally power down their computer), the module update manager 202 detects this and takes appropriate action. For example, the process of installing entwined updates will not take place until all updates have been downloaded. If an update is interrupted, it can be resumed at a later time. Only when all modules have been successfully downloaded (particularly entwined updates), will the updates be installed. This assures that at no time will a product be "partially updated" and inoperable.

[0073] As each update file is downloaded, its CRC is checked against the central database which is stored in server 2 and which contains (among other things) the CRC value of every asset, module, and update that can be downloaded. If the file was corrupted during download, the module update manager 202 recognizes this and attempts to download it again and/or allow the user the option of trying later. If a single member of an entwined set is corrupted, no members of the set will be installed.

[0074] Referring again to Fig. 5, when the game platform is started by the customer as indicated at 220 a check is made at process block 222 of all critical platform runtime code modules. This includes a CRC check and it also detects if any modules have had viruses attached to them. In either case, if corrupted critical platform modules are identified, the customer is alerted and given the choice as indicated at decision block 224 to update them using the module update manager 202. After any such updates are completed, the game program indicated generally at 226 is performed.

[0075] The first function performed by the game program is a call to initialize the platform as indicated at process block 228. This call tells the platform that it is about to run a product. The platform then performs initialization procedures indicated at process block 230, including setting up a module loader 232. The assets and modules needed by the game are then loaded into RAM by the module loader 232. Although most will load at startup, the developer has control over this process so that if memory conservation is a factor, runtime code modules can load/unload "sections" of the game on an as-needed basis. Regardless of when an asset or a module is loaded, the loader 232 CRC checks the asset or module during the load to assure that it has not been corrupted or tampered with. Should a module fail this CRC test, the user is given the option of downloading an uncorrupted version from the server 2 at decision block 234. Note that any copy protection modules used by the game developer are also checked at this time. Should an end-user attempt to "hack" around any copy protection or shareware registration schemes, the program will not operate.

[0076] The main game code indicated at process block 236 is then performed. As discussed above, this runtime code is primarily level 4 code which makes calls to the local module database 200 in the game platform to perform levels 1-3 functions. This is the "normal operation" of the game program and it continues to operate in this manner until the user ends the game at 238.

[0077] When the user ends the game, a platform shutdown procedure 240 is called. It releases all allocated memory, closes any open file handles, and unloads any modules that were in use. This "cleanup" process leaves the computer as it was before the game was executed, and assures that all modules are unloaded so that if the user were to immediately run an older game requiring older modules, would not be any backward-compatibility issues.

[0078] When the user quits a game he/she is returned to the game platform welcoming page. The user can select a different game to play or quit the game platform entirely. When the game platform is exited, the computer is transformed back to the computer operating system to again become a general purpose computer.

CLAIMS

1. A game platform for a general purpose computer which comprises:

means for storing a local module database containing a set of low level program modules which direct hardware elements of the general purpose computer to perform functions associated with games;

means for receiving a game program and updating the local module database with updated program modules associated with the game program; and

means for performing the game program using both low level program modules and updated program modules stored in the local module database;

whereby the platform transforms the general purpose computer into a single purpose game computer.

2. The game platform as recited in claim 1 which includes:

means for determining what program modules are required to perform the game; and

means for comparing the required program modules with the contents of a stored installed-module-table to determine what updated program modules are to be added to the local module database.

3. The game platform as recited in claim 2 in which the game program is stored on a removable media which is inserted into the general purpose computer, and the means for determining what program modules are required includes means for reading a required-module-table stored on the removable media.

4. A method for making a general purpose computer operate like a special purpose game machine for performing game programs, the steps comprising:

a) storing a local module database containing a set of low level program modules which direct hardware elements of the general purpose computer to perform a corresponding set of functions associated with games; and

b) installing a game platform program which performs a game program by directing the low level program modules stored in the local module database;

whereby games which use functions performed by the stored low level program modules may be performed by the general purpose computer without performing a program installation procedure.

5. A game program development system which comprises:

a server for storing a master asset pool containing program segments for performing computer game functions;

a set of workstations coupled to the server, each workstation being operable to create and edit game program segments and store them in the master asset pool; and

project builder means for producing a game program by concatenating game program segments stored in the master asset pool and combining the game program with a game platform program, that when installed in a general purpose computer, enables the general purpose computer to play the game program.

6. The game program development system as recited in claim 5 in which each workstation is provided with a different set of development tools that are tailored to the needs of different members of a game development team.

7. The game program development system of claim 5 in which a set of tools are tailored for a visual artist and the game program segments produced by this set of tools are images.

8. The game program development system of claim 7 in which a second set of tools are tailored for a sound artist and the game program segment produced by this set of tools are sound files.

9. The game program development system of claim 8 in which a third set of tools are tailored for the game developer and the game program segments produced by this set of tools are scripts.

10. A game program development system which comprises:

a server having memory for storing a master asset pool comprised of assets used in a game program;

a work station coupled to the server and having a memory for storing a local asset pool comprised of assets used in a game program;

a development tool used at the work station to create an asset and import the asset into the local asset pool;

a test program operable at the work station to test a game program using the local asset pool; and

an update controller operable at the work station to transfer the created asset from the local asset pool to the master asset pool.

11. The game program development system as recited in claim 10 which includes a test program operable at the server to test a game program using the master asset pool.

12. The game program development system as recited in claim 11 in which the update controller includes means for transferring assets from the master asset pool to the local asset pool.

13. The game program development system as recited in claim 10 in which there are a plurality of work stations, each having a memory for storing a local asset pool, each having a development tool for creating assets for their respective local asset pools, each having a test program operable to test a game program using their respective local asset pools, and each having an update controller operable to transfer created assets to the master asset pool.

14. The game program development system as recited in claim 13 in which the development tool at each of the plurality of work stations is different from each other.

15. The game program development system as recited in claim 10 in which the master asset pool is divided into sections, each of which includes assets associated with a common location in the game.

16. The game program development system as recited in claim 10 in which data is stored in the server memory which indicates for each asset in the master asset pool whether it is language independent or whether it is localized to a particular language.

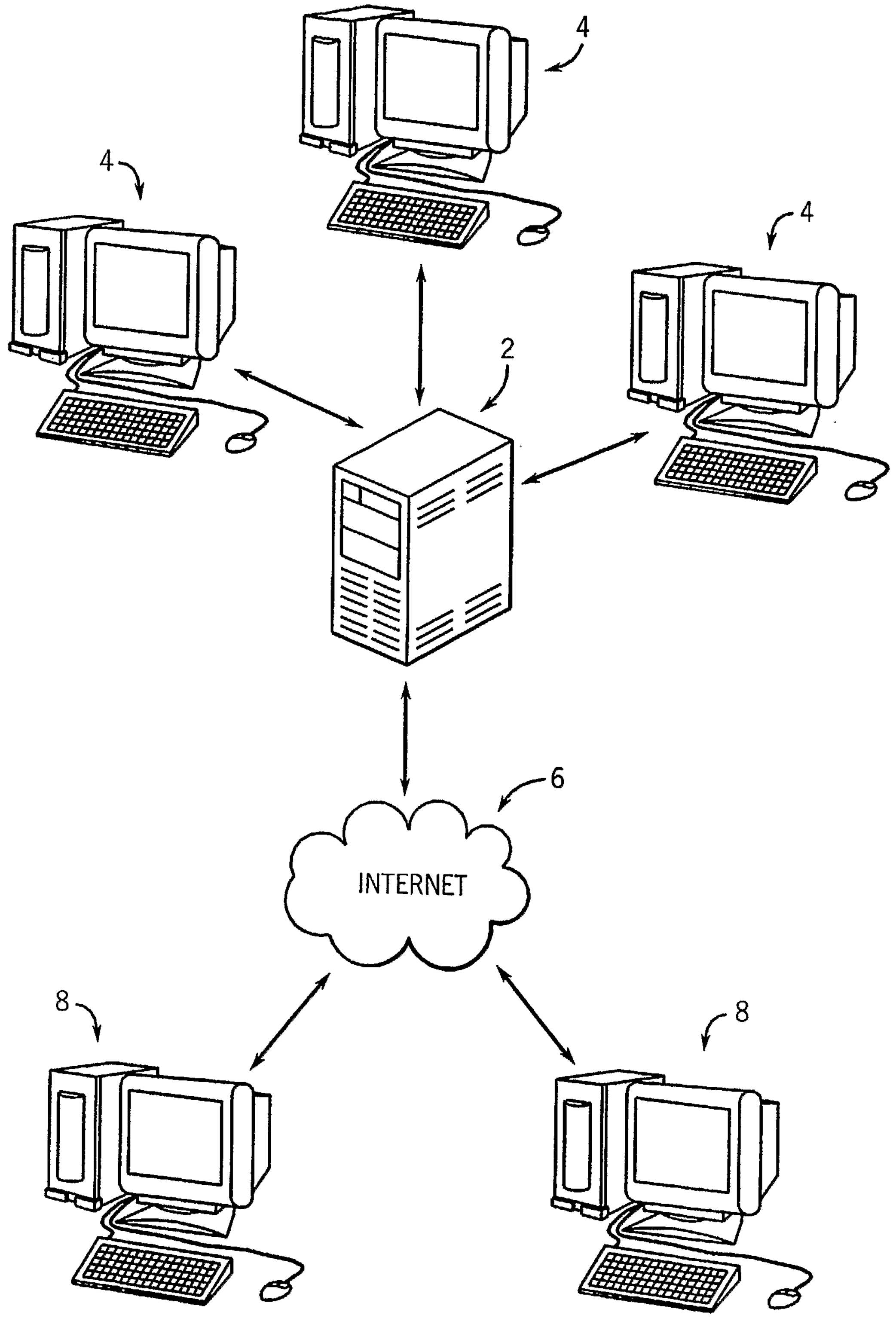


FIG. 1

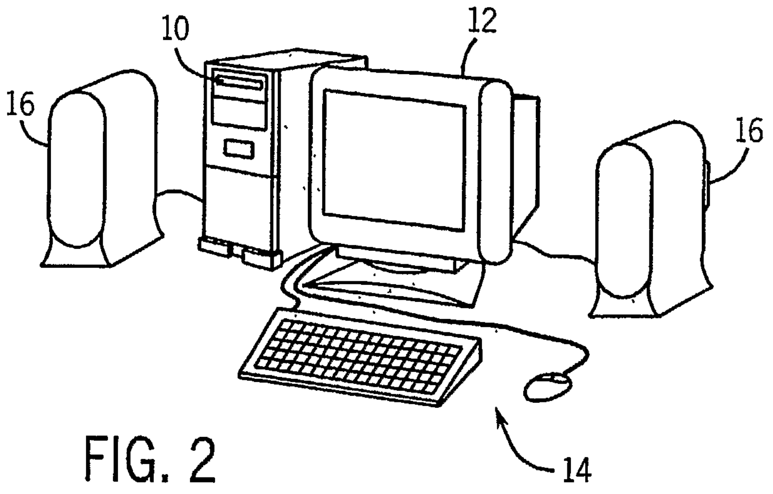

10
12
16
16
14

FIG. 2

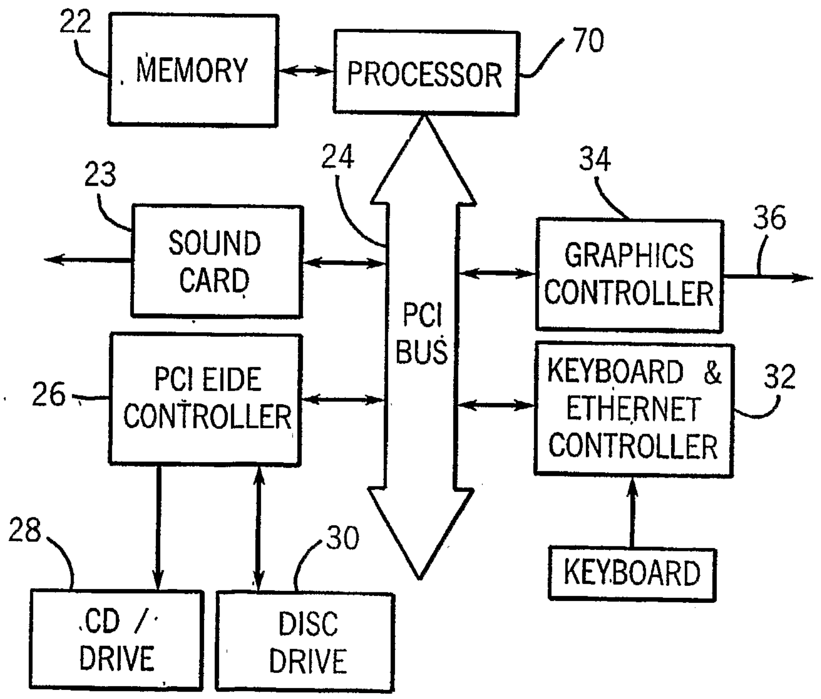

22
MEMORY
PROCESSOR
70
24
23
SOUND CARD
PCI BUS
34
GRAPHICS CONTROLLER
36
26
PCI EIDE CONTROLLER
KEYBOARD & ETHERNET CONTROLLER
32
28
30
KEYBOARD
CD / DRIVE
DISC DRIVE

FIG. 3

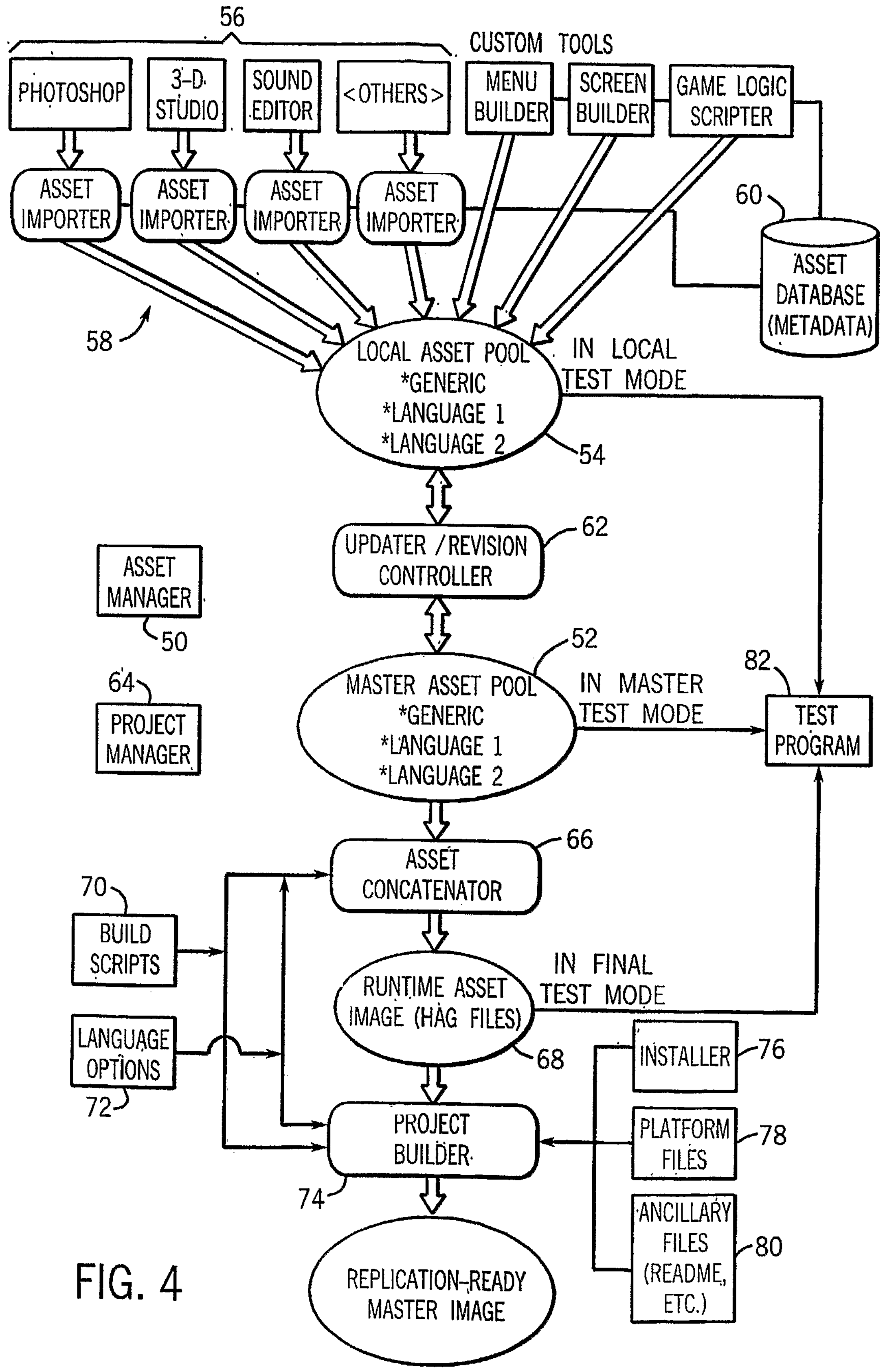

56
CUSTOM TOOLS
PHOTOSHOP
3-D STUDIO
SOUND EDITOR
<OTHERS>
MENU BUILDER
SCREEN BUILDER
GAME LOGIC SCRIPTER
ASSET IMPORTER
ASSET IMPORTER
ASSET IMPORTER
ASSET IMPORTER
60
ASSET DATABASE (METADATA)
58
LOCAL ASSET POOL *GENERIC *LANGUAGE 1 *LANGUAGE 2
IN LOCAL TEST MODE
54
UPDATER / REVISION CONTROLLER
62
ASSET MANAGER
50
64
PROJECT MANAGER
52
MASTER ASSET POOL *GENERIC *LANGUAGE 1 *LANGUAGE 2
IN MASTER TEST MODE
82
TEST PROGRAM
ASSET CONCATENATOR
66
70
BUILD SCRIPTS
RUNTIME ASSET IMAGE (HAG FILES)
IN FINAL TEST MODE
68
LANGUAGE OPTIONS
72
INSTALLER
76
PLATFORM FILES
78
PROJECT BUILDER
74
ANCILLARY FILES (README, ETC.)
80
REPLICATION-READY MASTER IMAGE

FIG. 4

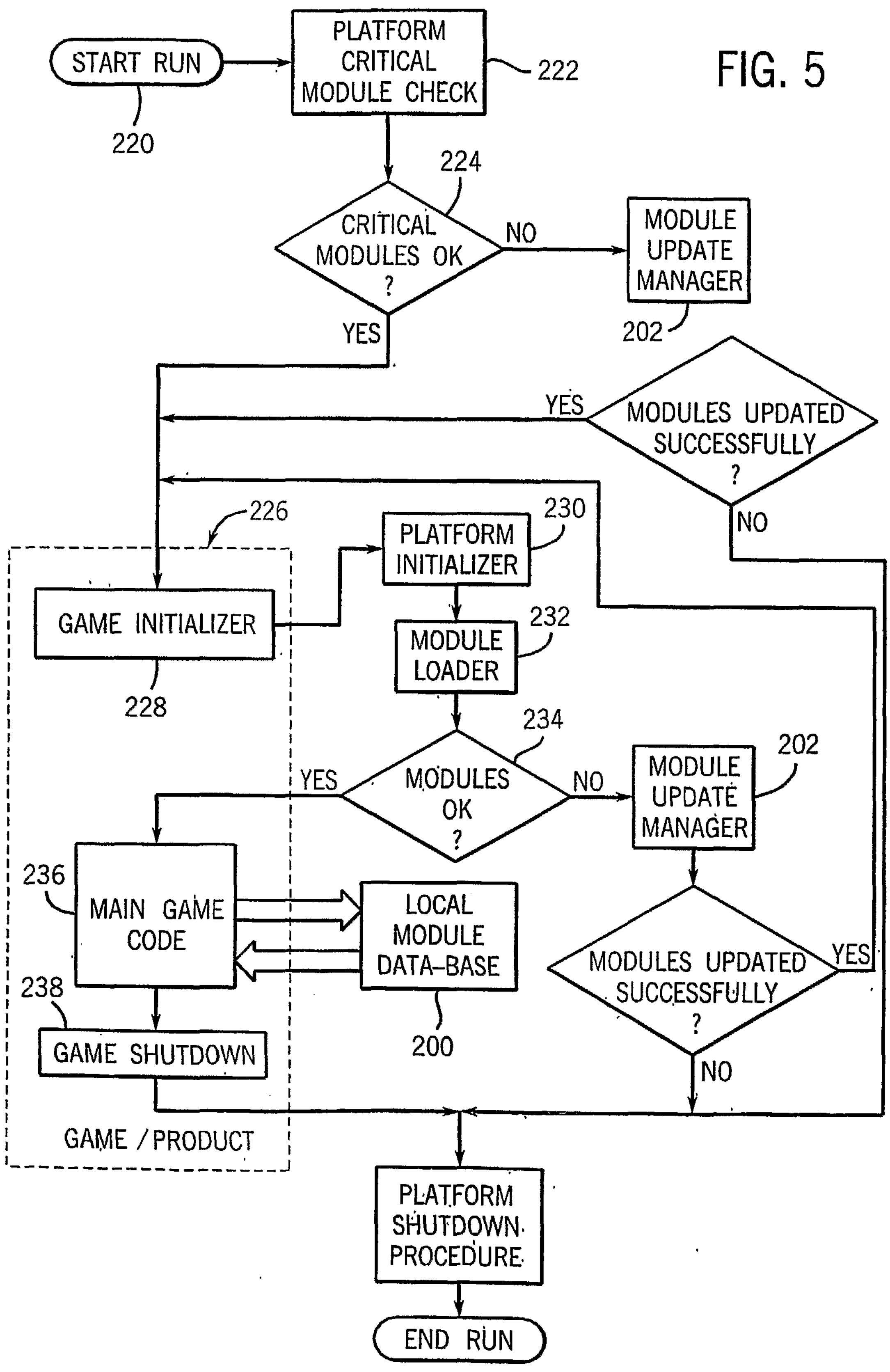

START RUN
220
PLATFORM CRITICAL MODULE CHECK
222
CRITICAL MODULES OK ?
224
NO
YES
MODULE UPDATE MANAGER
202
MODULES UPDATED SUCCESSFULLY ?
YES
NO
226
GAME INITIALIZER
228
PLATFORM INITIALIZER
230
MODULE LOADER
232
MODULES OK ?
234
YES
NO
MODULE UPDATE MANAGER
202
MAIN GAME CODE
236
LOCAL MODULE DATA-BASE
200
MODULES UPDATED SUCCESSFULLY ?
YES
NO
GAME SHUTDOWN
238
GAME / PRODUCT
PLATFORM SHUTDOWN PROCEDURE
END RUN

FIG. 5

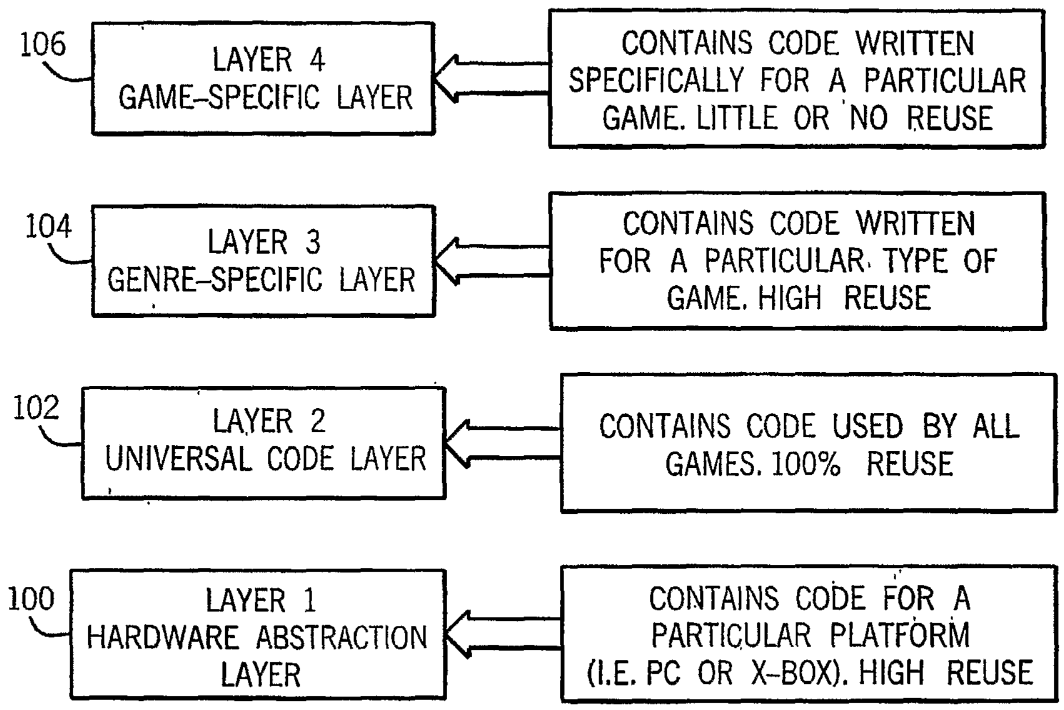

106
LAYER 4
GAME-SPECIFIC LAYER
CONTAINS CODE WRITTEN SPECIFICALLY FOR A PARTICULAR GAME. LITTLE OR NO REUSE
104
LAYER 3
GENRE-SPECIFIC LAYER
CONTAINS CODE WRITTEN FOR A PARTICULAR TYPE OF GAME. HIGH REUSE
102
LAYER 2
UNIVERSAL CODE LAYER
CONTAINS CODE USED BY ALL GAMES. 100% REUSE
100
LAYER 1
HARDWARE ABSTRACTION LAYER
CONTAINS CODE FOR A PARTICULAR PLATFORM (I.E. PC OR X-BOX). HIGH REUSE

FIG. 6

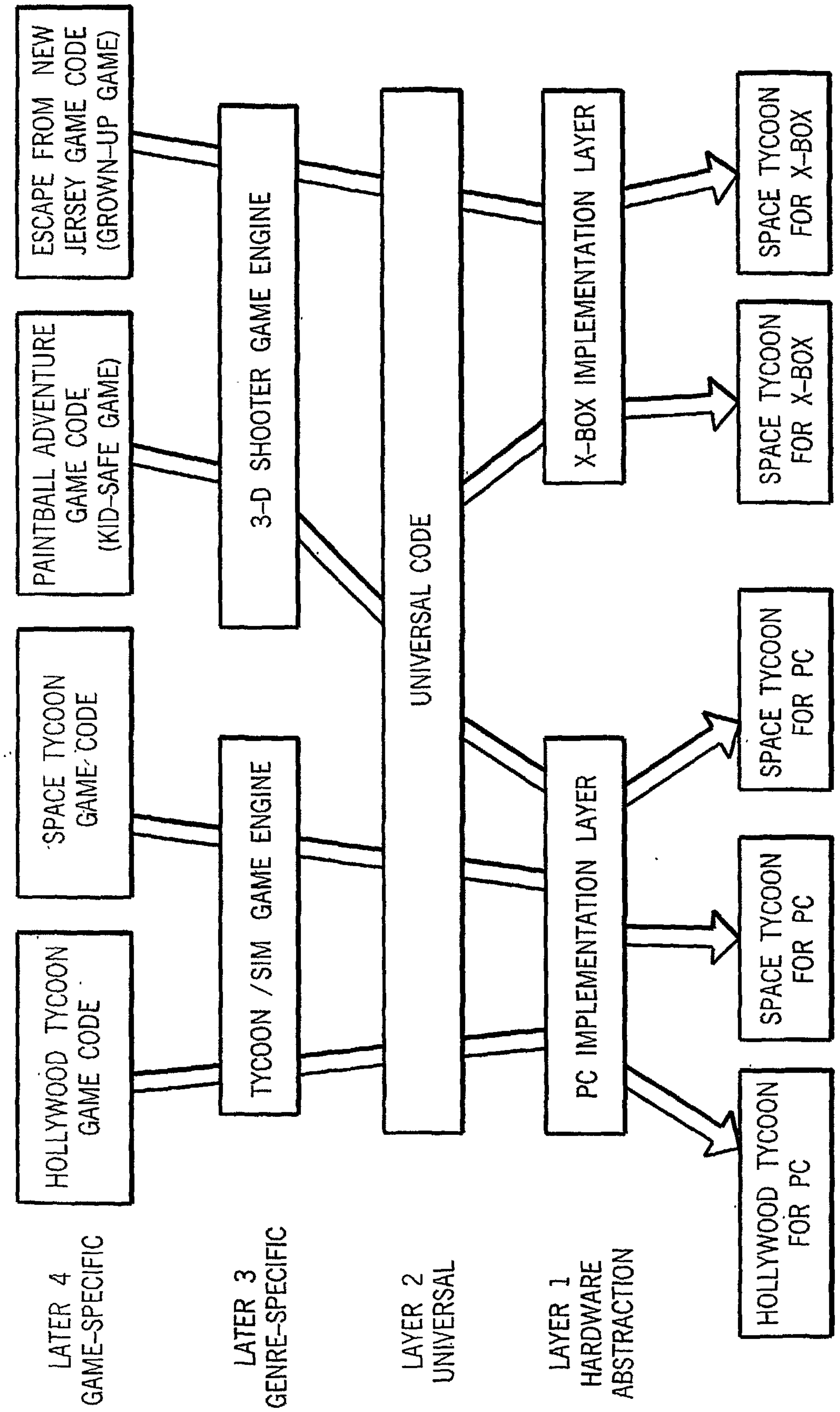

LATER 4 GAME-SPECIFIC
LATER 3 GENRE-SPECIFIC
LAYER 2 UNIVERSAL
LAYER 1 HARDWARE ABSTRACTION
HOLLYWOOD TYCOON GAME CODE
SPACE TYCOON GAME CODE
PAINTBALL ADVENTURE GAME CODE (KID-SAFE GAME)
ESCAPE FROM NEW JERSEY GAME CODE (GROWN-UP GAME)
TYCOON /SIM GAME ENGINE
3-D SHOOTER GAME ENGINE
UNIVERSAL CODE
PC IMPLEMENTATION LAYER
X-BOX IMPLEMENTATION LAYER
HOLLYWOOD TYCOON FOR PC
SPACE TYCOON FOR PC
SPACE TYCOON FOR PC
SPACE TYCOON FOR X-BOX
SPACE TYCOON FOR X-BOX

FIG. 7

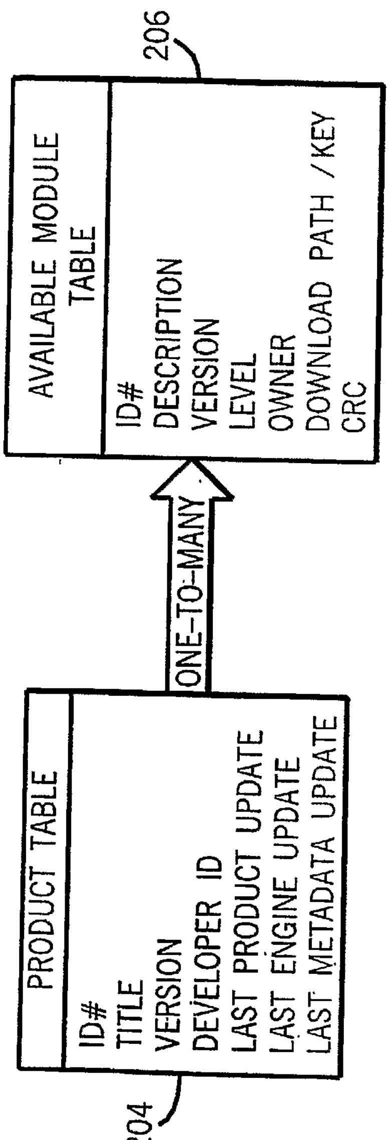

PRODUCT TABLE
ID#
TITLE
VERSION
DEVELOPER ID
LAST PRODUCT UPDATE
LAST ENGINE UPDATE
LAST METADATA UPDATE
204
ONE-TO-MANY
AVAILABLE MODULE TABLE
ID#
DESCRIPTION
VERSION
LEVEL
OWNER
DOWNLOAD PATH /KEY
CRC
206

FIG. 8

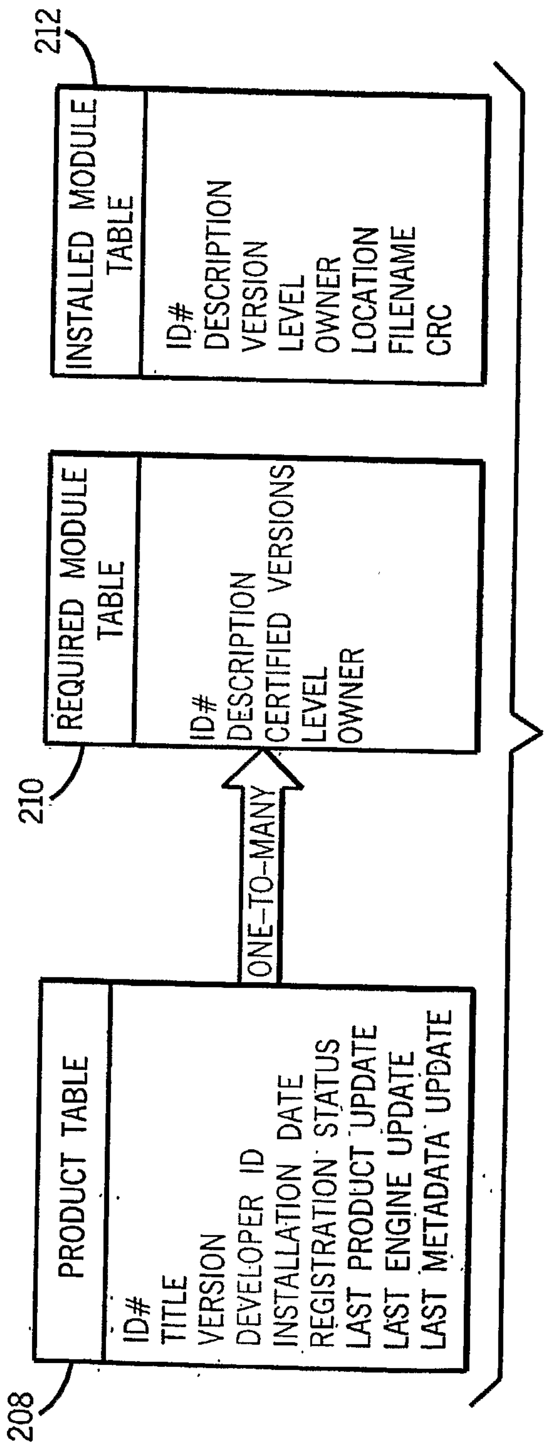

PRODUCT TABLE
ID#
TITLE
VERSION
DEVELOPER ID
INSTALLATION DATE
REGISTRATION STATUS
LAST PRODUCT UPDATE
LAST ENGINE UPDATE
LAST METADATA UPDATE
208
ONE-TO-MANY
REQUIRED MODULE TABLE
ID#
DESCRIPTION
CERTIFIED VERSIONS
LEVEL
OWNER
210
INSTALLED MODULE TABLE
ID#
DESCRIPTION
VERSION
LEVEL
OWNER
LOCATION
FILENAME
CRC
212

FIG. 9