Dokumentation

Projekt:

Netzwerkstar 2

Einleitung

Das vorliegende Projekt ist als online abrufbares Browser-Spiel ausgelegt. Es wurde mit Hilfe der Programmiersprache Javascript erstellt und nutzt den Hypertext Markup Language-Standard (HTML) in Version 5 zur Ausgabe. Darüber hinaus wird auf CSS-Formatierung und –Animationen zurückgegriffen.

Alle das Spiel betreffenden inhaltlichen Daten werden extern bereitgestellt. Zu diesem Zweck wurden alle Programmteile möglichst flexibel ausgelegt. Inhaltliche Änderungen erfordern deshalb keine großen Eingriffe in den Programmcode.

Konzepte

Die Spielinhalte werden extern bereitgestellt und müssen dem Programm auf zwei Arten zur Verfügung gestellt werden.

Alle optischen Inhalte liegen dazu als Bilddateien vor. Für diesen Zweck hat sich das Portable Network Graphics-Format (PNG) als am besten geeignet herausgestellt. Obwohl damit keine minimale Dateigröße erreicht wird, bietet es gegenüber dem Joint Photographics Experts Group File Interchange-Format (JPEG) die Möglichkeit, transparente Bildbereiche zu definieren. Das Projekt nutzt dennoch beide Formate, falls diese Funktion nicht benötigt wird.

Die übrigen Inhalte, wie zum Beispiel der Verlauf der Geschichte und die Dialoge der Charaktere, werden in Extensible Markup Language-Dateien (XML) hinterlegt. Hierbei ist auf die spezielle Formatierung und den Aufbau der Dateien zu achten. Dadurch lassen sich viele Funktionen und Eigenschaften der Inhalte im späteren Spiel beeinflussen.

Generell erfolgt eine Unterteilung der Handlung in einzelne Szenen. Diese beinhalten jeweils mehrere Dialoge zwischen den Figuren und weiterhin fest definierte Aufgaben, die durch den Spieler zu lösen sind.

Nach dem Aufruf des Spiels über einen Browser werden die Spielinhalte geladen. Dazu muss der Browser zwingend das Nachladen von Inhalten per Javascript und die Darstellung von HTML5 unterstützen.

Das Spiel beginnt mit einem Auswahlmenü in dem ein neues Spiel gestartet, ein vorheriger Stand geladen oder die Homepage aufgerufen werden kann. Nach dem Start oder dem Ladevorgang wird die entsprechende Szene angezeigt.

Die Darstellung erfolgt dabei zweidimensional. Durch die Anordnung der Inhalte kann jedoch ein dreidimensionaler Eindruck erweckt werden. Dabei werden insgesamt 5 Ebenen genutzt. Neben dem Hintergrund existieren jeweils zwei Vorder- und zwei Hintergrundebenen für statische und dynamische Gegenstände im Spiel. Während statische Gegenstände keine weitere Veränderung erlauben, ermöglichen dynamische Gegenstände zum Beispiel Animationen. Die Protagonisten können zwischen den Ebenen wechseln und tragen damit zur Tiefenwahrnehmung bei.

Die genannten Animationen werden durch Tilesets realisiert. Dabei handelt es sich um Standbilder, die mehrere kleine Einzelbilder einer Animation beinhalten. Durch das wiederholte Anzeigen dieser Einzelbilder entsteht der Eindruck einer Bewegung.

Dateiorganisation

Alle Spieldateien befinden sich in einem Verzeichnis. Auf der obersten Ebene sind definierenden XML-Dateien für die Spielszenen und die Dialoge abgelegt. Alle Bilder befinden sich im entsprechend benannten Ordner. Er beinhaltet die dritte XML-Datei, in der alle Bilddateien hinterlegt und mit

Zusatzinformationen versehen sind. Die Daten sind dabei in szenenspezifischen Unterordnern organisiert. Die beinhaltende HTML-Seite befindet sich im Unterordner „html“ des Wurzelverzeichnisses. Hier ist auch die CSS-Definitionsdatei gespeichert. Der Unterordner „js“ stellt alle Javascript-Dateien bereit. Diese sind noch einmal unterteilt. Im Ordner „external“ befindet sich die Bibliothek jQuery in der Version 1.8.3. Sie stellt viele verschiedene Komfortfunktionen bereit und wurde nicht im Rahmen dieses Projekts entwickelt. Der Ordner „core“ beinhaltet alle spielspezifischen Programmdateien.

Programmablauf

Nach dem Aufruf der HTML-Seite wird zuerst eine Animation angezeigt. Ihr Inhalt ist innerhalb der HTML-Datei definiert. Der Text wird unter Nutzung von CSS-Animationen dargestellt. Die entsprechenden Parameter befinden sich in der CSS-Definition. Für eine möglichst weitreichende Unterstützung verschiedener Browser wurden mehrere vorläufige CSS-Eigenschaften implementiert. Die entsprechenden Standards sind noch nicht in alle gängigen Programmversionen aufgenommen worden.

Nach einem Klick auf „Neues Spiel“ beginnt die Verarbeitung der Spieldaten mit dem Laden der XML-Dateien. Hierzu beinhaltet das Programm zwei Parser. Sie starten den Ladevorgang und die Verarbeitung, sobald die jeweilige Datei vollständig übertragen wurde.

Bilder

Der *pictureParser* verarbeitet die *bilder.xml*-Datei. Durch den Aufruf der Methode *ladeBilder* wird die Datei heruntergeladen. Mithilfe des Parameters force_load_common lässt sich ein Laden der Bilder im Ordner *allgemein* erzwingen. Dieser Ordner wird für szenenübergreifende Inhalte wie zum Beispiel die Protagonisten verwendet.

In der ersten Szene steht der Parameter deshalb auf *true*. Der Inhalt der Datei wird an *verarbeiteBilderXML* übergeben. Hier bestimmt die globale Variable *gcurrent_scene_counter* über die Filterung der Daten. Nur die für die aktuelle Szene relevanten Inhalte werden erfasst.

Alle globalen Variablen sind in der Datei *helper.js* definiert. Sie stellt den zentralen Speicher für verschiedene Klassen, Methoden und Variablen dar.

Die Filterung erfolgt mithilfe des *id*-Parameters. Er ist für jedes Bild in der *bilder.xml*-Datei definiert und identifiziert es eindeutig. Nachdem alle relevanten Bilder so erfasst wurden, wird ihre Anzahl im globalen Objekt *gBilder* gespeichert. Sie dient später der Überprüfung, ob alle Inhalte erfolgreich geladen werden konnten. Ähnliches gilt für das Flag *gpictureparser_xml_geladen*, welches den erfolgreichen Ladevorgang der XML-Datei signalisiert.

Das Laden der Bilddaten selbst erfolgt in der Methode *verarbeiteBilder*. Ihr wird die vorher erstellte Liste von Bildern übergeben. Innerhalb der Methode werden die weiteren XML-Attribute ausgelesen und jedes Bild als Instanz der Klasse *Bild* in *gBilder* abgelegt. Der Zugriff erfolgt dabei später über die genannte Bild-id. *Bild* speichert neben der *id* auch den Pfad und die Abmessungen des Bildes, wie sie in der XML-Datei hinterlegt sind. Falls das Bild animiert werden soll, wird das entsprechende Flag auf *true* gesetzt und zusätzlich eine Instanz der Klasse Animationsmerkmale mit den Attributen *fps*, *tile_anzahl*, *tile_width* und *tile_height* erzeugt. Sie beinhaltet die nötigen Angaben für die Darstellungen der Bildanimation.

Das instanziieren der Klasse *Bild* bewirkt den Start des Ladevorgangs. Gleichzeitig wird eine Funktion definiert, welche nach dem vollständigen Vorgang ausgeführt wird. Sie ruft unter anderem die Methode *waitforparser* in *helper.js* auf und erhöht den Zähler für geladenen Bilder in *gBilder*.

Dialoge

Der *dialogParser* arbeitet sehr ähnlich wie das Pendant für Bilder. Die Methode *verarbeiteDialogeXML* prüft mithilfe von *gInitialLoad*, ob es sich um den Beginn einer Szene handelt. In diesem Fall wird das globale Array *gDialogIDs* angelegt. Andernfalls wird dessen Inhalt in den Variablen *gDeprecatedDialogIDs* und *gBackupOfDialogs* gespeichert, bevor es überschrieben wird. Dies passiert kurz vor dem Ende einer Szene, wenn bereits die Inhalte der Folgenden vorgeladen werden. *gUseDeprecatedDialogues* wird dann auf *true* gesetzt und bewirkt damit, dass alle alten Dialoge der endenden Szene in *gDeprecatedDialogues* überführt und bis zum Beginn des nächsten Abschnitts von dort gelesen werden. *gDialoge*, als Behälter für die Szenentexte, wird damit frei für neue Inhalte. Diese werden anhand der aktuellen Szenennummer aus den XML-Daten gefiltert und dann, wie im *pictureParser*, in einzelnen Instanzen der Klasse *Dialog* gespeichert. Sie stellt später die Texte zur Anzeige anhand ihrer *id* bereit. Für die Verwendung in den Spielrätseln wird separat das Array *gDialogIDs* mit *DialogIDObject*-Instanzen gefüllt. Sie speichern die XML-Attribute, die über die Auswirkungen des Dialogs auf den Fortschritt im Spiel entscheiden. Dabei bestimmt *increase_quiz_step* darüber, ob dieser Dialog das Rätsel vorantreibt, *trigger_at_start* bewirkt eine Anzeige direkt nach Szenenbeginn ohne Spielertätigkeit und *invoke_scene_exception* stellt mit *argument_list* die Möglichkeit bereit, Ausnahmebehandlungen zu definieren.

Dabei handelt es sich um Funktionen, die in *sceneExceptioins.js* hinterlegt werden können. Sie dienen der Ausführung von Sonderfunktionen in speziellen Situationen und bieten so beliebige Erweiterungsmöglichkeiten für das gesamte Programm. i*nvoke_scene_exception* wirkt dabei gleichermaßen als Flag, als auch als Identifizierung der gewünschten Funktion, während *argument_list* die zugehörigen Parameter bereitstellt. Beide lassen sich mit dem Wert *#none#* deaktivieren.

Jeder Dialog der per *increase_quiz_step* entsprechend markiert wurde, erhöht den Zähler *gQuizTrueQuizSteps*. Da zwischen Rätselschritten und Fortschritt in der Szene unterschieden wird, ist dieser Punkt wichtig, denn nur wenige Dialoge treiben das Rätsel voran.

Jeder Dialog kann ein Gespräch zwischen mehreren Teilnehmern abbilden. Deshalb ist die XML-Datei in einzelne Sätze unterteilt und das *Dialog*-Objekt speichert diese in einem Array aus *Satz*-Instanzen. Neben dem Text definiert jeder Satz auch den Sprecher und dessen Repräsentation im Dialogfenster. Die *bild_id* ist dabei eine Referenz auf *gBilder*, welches das entsprechende Bild bereitstellt.

Wie zuvor die Bilder bewirkt die Instanziierung der Klasse *Dialog* den Aufruf von *waitforparser*.

Szene

Nachdem alle Inhalte geladen wurden, erfolgt die Anzeige. Dazu prüft die Methode *waitforparser* bei jedem Aufruf den Status der einzelnen Parser. Erst wenn alle Bilder und Dialoge geladen wurden, erfolgt der Aufruf von *getSceneInformation* mit der aktuellen Szenennummer und der zu ladenden XML-Datei. Dies geschieht allerdings nur wenn *gdisplay_next_scene* auf *true* steht, was bei der ersten Szene der Fall ist. Nachdem eine Szene durch *sceneParser.js* geladen wurde, wird die Variable umgeschaltet und erst nach dem Durchlaufen des Rätsels durch *quizControl.js* wieder zurückgesetzt.

getSceneInformation legt zu Beginn eine Instanz von *sceneStruct* an. Die Klasse beinhaltet später alle Objekte der Szene und speichert diese in fünf Arrays, eines für jede Bildebene. Die aktuelle Szene wird aus der Datei *szenen.xml* eingelesen. Jede Szene definiert ein Attribut *rätselschritte*. Es wird global in *gQuizsteps* gespeichert und dient der Überprüfung des Fortschritts. Sobald der Spieler hier den vorletzten Schritt erreicht, wird das Laden der nächsten Szene ausgelöst. Die Inhalte werden dann bei Erreichen des letzten Schrittes angezeigt.

Da einige Szenen im Weltraum spielen und dies Auswirkungen auf die Darstellung der Protagonisten hat, wird das Attribut *spc* ebenfalls ausgewertet. Für die betreffenden Szenen ist es in der XML-Datei auf *true* gesetzt.

Als *wegpunkt* sind in der Datei die Bildschirmkoordinaten gespeichert, die die Protagonisten auf dem Weg zu Spielobjekten ansteuern. Die Angaben für *x* und *y* stellen dabei relative Koordinaten innerhalb des Browser-Fensters dar und *zoom* bestimmt über die Ebene in der die Figur an der Position dargestellt wird. Ihre Größe wird dementsprechend angepasst. Der Parameter beeinflusst auch die Darstellung aller anderen Spielobjekte und bewirkt maßgeblich den Tiefenwahrnehmungseffekt. An diesem Punkt werden die Arrays *gWegPos* mit den Koordinaten und *gZoomsteps* mit einzelnen Zoomschritten gefüllt.

Die Objektebenen werden mithilfe der Methode *getSceneElementData* ausgelesen. Ihr werden alle zugeordneten XML-Elemente übergeben. Sie beinhalten die *id* des anzuzeigenden Bildes, die *id* des verknüpften Dialog-Objektes, die Rätsel-Flags sowie Positions- und Größenangaben. Letztere sind wiederum relative Angaben bezogen auf die Fensterabmessungen. Im Unterschied zu den Wegpunkten repräsentiert die z-Position jedoch den z-Index im Sinne einer CSS-Eigenschaft. Während Objekte ihre Position nicht ändern, wird dieser Wert für Personen dynamisch angepasst, wenn sie sich bewegen.

Zunächst wird ein temporäres *objectStruct*-Objekt erzeugt. Es speichert die genannten *id*-Attribute und alle Quizinformationen. Zusätzlich wird jedes Spielobjekt in *gImageToObjectSceneReferrer* aufgenommen. Dort werden die angezeigten Bilder mit den Dialogen verknüpft, welche ihnen durch die XML-Datei zugeordnet sind. Auf diese Weise muss nicht in den verschiedenen Datenbehältern nach den Objekten gesucht werden wenn der Spieler etwas anklickt.

Dem temporären Objekt werden noch die Rätsel-Flags hinzugefügt, bevor es für die weitere Auswertung an *getElementData* übergeben wird. Die Flags bestimmen neben der Sichtbarkeit auch darüber, ob ein Objekt das Rätsel vorantreibt, für den Spieler anklickbar ist und ob die Spielfiguren zu seiner Position gehen können. Die Information ist dabei als Array aus booleschen Werten kodiert. Jedes „f“ entspricht *false* und jedes „t“ dem Wert *true* für den Rätselschritt, der seinem Index im Array entspricht. Das Array beinhaltet dabei immer dieselbe Anzahl Einträge, die unter *reatselschritte* definiert wurde.

Innerhalb von *getElementData* wird neben der Position und Abmessung eines Objektes auch zusätzlich das optionale Attribut *laufziel* ausgelesen. Es definiert abweichende Koordinaten, die ein Charakter ansteuern kann. Ohne das Attribut wird stets die Mitte der unteren Kante eines Objektes als Ziel der Laufanimation gesetzt. Die Methode *getPersonElementData* wird für alle beschriebenen *person*-Elemente der XML-Datei aufgerufen. Sie lässt die Rätselattribute aus, funktioniert aber sonst analog zu *getSceneElementData*.

Nachdem jedes Element ausgelesen und dem entsprechenden Array innerhalb des *sceneObject* hinzugefügt wurde, kann die Methode *drawScene* mit diesem Objekt aufgerufen werden. Sie übernimmt die Darstellung der Szene. Anschließend wird *gdisplay_next_scene* auf *false* gesetzt, um ein Laden der nächsten Szene vor den zugehörigen Inhalten zu verhindern. Die Funktion *HideElementsMenu* blendet das Spielmenü aus und mithilfe von *scene1_hideHeroine* wird die Protagonistin in der ersten Szene ausgeblendet. Die Methode ist ein Beispiel für die Ausnahmebehandlungen in *sceneExceptions.js*.

sceneNeedsForcedDialog sucht im Array *gForceDialogScenes* nach Szenen deren Handlung die Anzeige eines Dialogs zu Beginn vorsieht. In diesem Fall wird durch die Methode *forceDialog* in

dialogControl.js der definierte Dialog angezeigt. Dies geschieht durch die Methode *dialog_zeichneDialog*.

Nachdem dieser Dialog, wenn nötig, angezeigt wurde, wird in *sceneParser.js* noch der *gImageToObjectSceneReferrer* auf Objekte geprüft, welche nicht zur aktuellen Szene gehören. Zur Vermeidung von Überschneidungen werden diese Objekte gefiltert, die sich beim Übergang zwischen Szenen ansammeln.

Abschließend erfolgt eine Prüfung, ob die Szene vom Spieler per Schlüsseleingabe geladen wurde. In diesem Fall setzt *advanceSceneToLastSavestate* das Spiel auf den geforderten Stand zurück. Die Methode ist in *codeGenerator.js* definiert und spielt alle nötigen Schritte zum Erreichen eines gegebenen Spielfortschritts ab. Dabei greift sie auf das *gCodegeneratorArray* zurück. Es beinhaltet für jede Szene ein Array mit Indexeinträgen entsprechend den Schlüsselwerten in *gClickEventValueArray*. Hier sind alle anklickbaren Objekte der Szene hinterlegt. Die Funktion emuliert den Klick auf diese Objekte und nutzt die vorhandene Logik zur Wiederherstellung des Spielstandes.

Das Zeichnen der Szene erfolgt pro Ebene in *drawScene*. Für jede Ebene wird *drawObjectsOfSameType* mit allen zuvor gesammelten Objekten, einem gemeinsamen Teil der ID und dem Flag *hasSingleCanvas* aufgerufen. Die Bilddaten der Objekte werden in der Methode in HTML *Canvas*-Elemente geschrieben. Diese sind unter einer ID erreichbar, welche sich unter anderem aus *sharedIdString* und dem *id*-Attribut des Bildes zusammensetzt. *hasSingleCanvas* bestimmt darüber, ob alle Objekte dieser Ebene stattdessen in ein einziges *Canvas*-Element gezeichnet werden. Dies trifft zum Beispiel für statische Hintergrundobjekte zu.

In jedem Fall für ein neues *Canvas*-Objekt erzeugt und dem HTML-Dokument hinzugefügt. Bei statischen Hintergrundobjekten nimmt dieses Element die Abmessungen des Fensters an. Anschließend werden alle zu zeichnenden Elemente nach ihrem z-Index sortiert. Für die Nutzung im Spielverlauf werden die wesentlichen Eigenschaften jedes Objekts an dieser Stelle innerhalb von *gImageStats* als Instanz von *imageStatObject* gespeichert. Dazu zählen die ID, Position und Größe sowie das separate l*aufziel*-Attribut des Objekts. Danach wird der Bildinhalt gezeichnet. Hierbei werden die relativen Angaben aus der XML-Datei durch *perc2pix* in *helper.js* in Bildschirmkoordinaten umgerechnet.

Abschließend wird die z-Index-Eigenschaft des *Canvas* gesetzt.

Der Ablauf für den Fall, dass *hasSingleCanvas* auf *false* oder nicht gesetzt wurde, ist sehr ähnlich aufgebaut. Wie bereits erwähnt erhält jedes *Canvas* eine eindeutige ID zusammengesetzt aus *sharedIdString* und *id*-Attribut des zugrundeliegenden Bildes. Danach erfolgt eine Unterscheidung zwischen der Personen- und den übrigen Ebenen.

Ebenenobjekte, die keine Person darstellen, erhalten weitere Merkmale, die ihrer ID hinzugefügt werden. Dazu zählen die *raetsel_sichtbar*- und *klickbar*-Flags aus der *szenen.xml*-Datei. Auf diese Weise sind die Informationen direkt verfügbar, wenn das *Canvas* angeklickt wird. Zusätzlich werden die *raetsel_ausloeser*- und *walkto*-Flags zusammen mit der ursprünglichen Canvas-ID und dem *id*-Attribut des zugehörigen Bildes in *gClickEventValueArray* gespeichert. Diese Struktur wird zum Beispiel beim Laden per Schlüsseleingabe und beim Klick auf ein Spielobjekt genutzt, um diese Informationen direkt verfügbar zu machen.

Jedem *Canvas* wird eine CSS-Klasse zugeordnet. Sie bestimmt die Sichtbarkeit des Objekts und den angezeigten Effekt wenn der Mauszeiger über dem Objekt schwebt. *raetsel_sichtbar* und *klickbar* entscheiden über das Hinzufügen der Klassen *quiz_hidden* oder *quiz_shown* und *clickable*. Zusätzlich

wird das *onclick*-Attribut des *Canvas* mit einem Funktionsaufruf versehen. Dadurch wird beim Klick darauf die Methode *startEventHandling* mit der ursprünglichen ID des Objektes aufgerufen.

Personen darstellende Objekte erhalten automatische die CSS-Klasse *quiz_shown* und keine weiteren *onclick*-Attribute.

Anschließend werden alle Ebenen wieder gleich behandelt. Die angegebenen Abmessungen werden in Bildschirmkoordinaten umgerechnet und unter Berücksichtigung der Skalierung angewendet. Diese wird mithilfe von *z2mult* in *helper.js* aus einem gegebenen z-Index berechnet und entspricht einem Wert innerhalb von *gZoomsteps*.

Die letzte Unterscheidung betrifft animierte Bilder. Statische Anzeigen werden direkt in das *Canvas*-Element gezeichnet. Animierte Bilder werden *pictureAnimation.js* und die Methode *starteAnimation* übergeben.

Animation

Wie bereits erwähnt werden animierte Bilder als Tilesets zur Verfügung gestell. Alle Einzelbilder der Animation sind darin gleich groß und in chronologischer Reihenfolge nebeneinander abgebildet. Verschiedene Animationen des gleichen Objektes lassen sich in einem Bild zusammenfassen. Dazu wird der „Filmstreifen“ um weitere Zeilen ergänzt.

Zum Starten einer Animation werden die ID des zu nutzenden *Canvas*, das Bild und die gewünschten Abmessungen in Pixel an *starteAnimation* übergeben. Daraufhin wird der entsprechende Eintrag in *gAnimationTimer* geprüft. Das Objekt enthält alle im Programmverlauf genutzten Animationen. Existiert hier kein Eintrag, wird eine neue Instanz der Klasse *Animation* erzeugt. Sie speichert die der Funktion übergebenen Parameter und macht sie so global verfügbar. Für den Fall, dass bereits eine Instanz mit derselben ID vorhanden ist, werden deren Abmessungen mit den neuen Werten überschrieben.

Derzeit werden verschiedene Animationen für ein Bild nur bei Spielfiguren unterstützt. Für alle übrigen Bilder steht der Parameter *subtileset* in *Animation* auf *0*. Andernfalls wird der Wert beim Erzeugen der Klasse automatisch auf *gInitialDirection* gesetzt. Die eigentliche Animation wird mithilfe von Zeitgebern umgesetzt. Diese rufen asynchron zum normalen Programmablauf regelmäßig die Methode *animiereCanvas* auf. Zur Steuerung wird die Timer-Nummer ebenfalls in der Klasse *Animation* abgelegt.

Das Intervall zwischen den Aufrufen wird aus *gBilder* gelesen. Hier wurde innerhalb von *sceneParser.js* für jedes betreffende Bild eine Instanz von *Animationsmerkmale* erzeugt. Sie enthält unter anderem die Anzahl der anzuzeigenden Bilder pro Sekunde. Je nachdem welchen booleschen Wert *gUseDeprecatedImages* annimmt, werden die Informationen aus *gDeprecatedImages* oder *gBilder* gelesen. Vor dem Ende einer Szene werden deren Inhalte verschoben und sind dann in den mit „deprecated“ benannten Objekten verfügbar.

Durch das Anlegen des Zeitgebers im System sind die Vorbereitungen für die Animation abgeschlossen. Der Wert *running* in *Animation* wird abschließend auf *true* gesetzt und der Zähler für die hinterlegten Einträge in *gAnimationTimer* inkrementiert.

Je nach angegebenem Intervall wird kurz darauf *animiereCanvas* aufgerufen. Hier wird zunächst die übergebene *Canvas*-ID von allen beigefügten Flags befreit und anschließend das zugehörige HTML-Element ermittelt. Danach wird dessen Zeichenfläche zunächst geleert und danach mit dem nächsten Bild in der Abfolge gefüllt. Zu diesem Zweck werden die Informationen aus *gBilder* und *gAnimationTimer* kombiniert und ein Ausschnitt des Tilesets, abhängig von den Abmessungen der

Einzelbilder und dem aktuellen Wert unter *subtileset*, ermittelt. Danach wird der Wert *bild_nr* in *gAnimationTimer* erhöht. Er bestimmt in jedem Durchlauf über das zu verwendende Einzelbild.

Während sich zum Beispiel die Figuren durch die Szene bewegen, passt sich ihre Blickrichtung der Bewegung an. Dieser Wechsel zwischen verschiedenen „Filmstreifen“ innerhalb des Tilesets erfolgt mithilfe der Funktion *switchWalkingAnimation*. Ihr wird neben der betreffenden Bild-ID zusätzlich die gewünschte Blickrichtung übergeben. Anschließend setzt die Funktion das Merkmal *subtileset* des betreffenden Bildes in *gAnimationTimer* auf den entsprechenden Wert. An dieser Stelle wird die Zuordnung dieses Wertes zu den „Filmstreifen“ des Tilesets realisiert.

Mithilfe von *stoppeAnimation* kann jede hinterlegte Bilderabfolge angehalten werden. Nach dem Stoppen des Zeitgebers werden die Werte *running* und *bild_nr* auf *false* beziehungsweise *0* gesetzt und der Animationszähler um eins verringert. Die Instanz der Klasse *AnimationK* bleibt dabei erhalten und steht für eine erneute Verwendung zur Verfügung.

Eingabeverarbeitung

Der Spieler kann jedes *Canvas*-Element im Fenster anklicken. Bis auf den Protagonisten und den Hintergrund, löst jedes Objekt die Ausführung der Funktion *startEventhandling* aus. Diese Funktion innerhalb von *clickEventHandler.js* dient als Ausgangspunkt für die folgenden Aufrufe. Ihr wird ein Schlüssel aus *gClickEventValueArray* übergeben.

Dieser Schlüssel wird zu Beginn in der globalen Variable *gMostRecentlyClickedIdentifier* hinterlegt. Anschließend wird der zugehörige Wert aus dem Array gelesen und in *parsedStringObject* gespeichert. Das Objekt enthält die aus dem Array ermittelten Werte. Diese werden dazu von der Funktion *parseValueString* aufbereitet, indem der enthaltende String in seine Bestandteile zerlegt wird. Anschließend stehen die ID des *Canvas*, dessen Rätsel-Flags und die zugehörigen Dialoginformationen in *parsedStringObject* zur Verfügung.

Nachdem die nötigen Informationen erfasst wurden, findet eine Überprüfung der gegebenen Voraussetzungen statt. Zunächst wird die Initialisierung des Dialogsystems geprüft. *gTalk* enthält dazu neben den voreingestellten Dialogeigenschaften das Flag *isInitialized*. Nur wenn es auf *false* steht, werden die Figuren bewegt und der Rätselfortschritt überprüft. Andernfalls wird derzeit ein Dialog angezeigt. Abhängig von *gEventHandlerBusy* erfolgen unterschiedliche Aktionen. Das Flag signalisiert eine derzeit ablaufende Bewegung der Spielfigur. Aus diesem Grund wird *bewegePerson* in *walkAnimation.js* nur bei einem Wert von *false* aufgerufen. Innerhalb der Funktion wird dann das Flag umgeschaltet und nach Abschluss der Bewegung zurückgesetzt.

Um zu verhindern, dass die ausgelesenen *Canvas*-Attribute überschrieben werden, wird *gQuizAndDialogArgumentsLocked* gesetzt. Während es auf *false* steht, werden die *quizFlags* aus *parsedStringObject* in *gQuizFlags* und die Informationen aus *dialogValue1* und *-2* in *gDialogValue1* und *-2* abgelegt. Damit werden die Werte im gesamten Programm global verfügbar. Dieser Schritt ist deshalb wichtig, weil viele Aktionen erst ausgelöst werden sollen, wenn der Protagonist sein angegebenes Ziel erreicht hat. Da Dialoge und Rätselschritte die Szene beeinflussen, wird ihre Verarbeitung hier verzögert und erst durch einen erneuten Aufruf von *startEventHandling* zu einem späteren Zeitpunkt ausgeführt. Gesteuert durch die Flags können dann die zuvor gespeicherten Informationen ausgewertet werden.

Am Ende der Funktion erfolgt eine Auswertung der ausgeführten Aktionen. Jeder Vorgang, der den Spielablauf voran gebracht hat, durch Bewegung der Figuren, Anzeige eines Dialogs oder Fortschritt im Rätsel, löst einen Aufruf von *codeGenerator.js* aus. Dadurch wird ein Schlüssel zum Laden des Spiels erzeugt.

Wie bereits erwähnt erfolgen einige Vorgänge verzögert nachdem die Spielfigur ihr Ziel erreicht hat. Dazu wird die Funktion *finishEventHandling* genutzt. Sie nutzt die global gespeicherten Daten und ruft *advanceQuizStep* und *advanceDialogStep* auf. Die Funktionen prüfen die übergebenen Parameter und führen das Rätsel fort oder eigen die entsprechenden Dialoge an. Abschließend wird *gQuizAndDialogArgumentsLocked* zurückgesetzt, sodass neue Werte für den nächsten Schritt gespeichert werden können.

Laufanimation

Zur Bewegung der Figuren wird *walkAnimation.js* verwendet. Die Datei beinhaltet die Methode *bewegePerson* die alle nötigen Berechnungen ausführt. Sie wird wiederholt aufgerufen und versetzt die Figur jeweils nur um einen Bruchteil des Weges, wodurch der Eindruck einer kontinuierlichen Bewegung entsteht.

Zu Beginn wird das Flag *gWegBerechnet* geprüft. Es signalisiert, ob die Funktion bereits aufgerufen und der Bewegungsvektor berechnet wurde. Außerdem werden die in *startEventHandling* hinterlegten Flags geprüft und die Bewegung nur ausgeführt, wenn in der *szenen.xml* das betreffende Attribut *walkto* für den Rätselschritt auf *true* gesetzt ist. Hier wird der Zusammenhang zwischen XML-Daten und Flags im Spiel deutlich.

Das angegebene Ziel der Bewegung wird in *gisWalkingTo* gespeichert, damit andere Funktionen währenddessen darauf zugreifen und die laufende Bewegung erkennen können. Danach werden die *Canvas* von Figur und Ziel erfasst. Die Skalierungsfaktoren beider Elemente werden mithilfe der Methode *z2mult* aus ihren z-Indizes berechnet und in *heroPos* und *targetPos* gespeichert. Diese Arrays enthalten die Bildschirmpositionen der *Canvas*-Elemente. Vor dem Beginn der Bewegung wird die Ausgangsgröße des Canvas mit dem Protagonisten gespeichert. Dessen Größe wird auf seinem Weg durch die Szene anhand dieser Daten angepasst. Hierbei gibt es zwei Möglichkeiten.

Die Figur geht in die Szene, wird also kleiner, oder kommt in den Vordergrund, weshalb sie größer werden muss. Der zweite Fall muss gleich zu Beginn von *bewegePerson* behandelt werden. Das Verhältnis zwischen den verschiedenen z-Indizes der *Canvas* wird dazu an *skaliereCanvas* übergeben. Die Funktion speichert den Inhalt des Elements und verändert seine Größe entsprechend. Der Inhalt wird danach wiederhergestellt, sodass sich optisch keine Änderung ergibt. Allerdings wurde die Zeichenfläche angepasst und bietet, im Falle einer Vergrößerung, mehr Raum.

An dieser Stelle wird die Dialogbox ausgeblendet. Sie soll während der Bewegung nicht sichtbar sein und ihr wird deshalb die CSS-Klasse „invisible“ zugewiesen. Danach wird die Position der Hauptfigur ausgelesen. Die CSS-Angaben werden dazu mit den zuvor gespeicherten Abmessungen kombiniert. Auf diese Weise wird ein Punkt berechnet, der mittig zwischen den Füßen des Helden liegt. Ähnliches gilt für das Bewegungsziel. Zunächst wird *gImageStats* nach einem passenden Eintrag für eine separate Zielangabe durchsucht. Wie bereits erwähnt, können diese Zielkoordinaten in der *szenen.xml* angegeben werden. Existiert kein Eintrag, wird die aktuelle Position des Zielobjekts ausgewertet und ebenfalls ein Punkt in der Mitte am unteren Ende des Objekts berechnet.

Als letzte Aktion vor der Wegberechnung wird noch einmal geprüft, ob die Figur sich bereits am Ziel befindet oder das Spiel gerade per Code geladen wird. In beiden Fällen wird die Bewegungsanimation sofort abgebrochen und *finishEventHandling* aufgerufen. Hierbei wird auch *gAktuellesZiel* berücksichtigt. Die Variable beinhaltet den aktuellen Index innerhalb von *gMoveVec* und zeigt an in welchem Bewegungsabschnitt sich die Animation gerade befindet.

Der Bewegungsvektor wird nun schrittweise berechnet. Zunächst werden die vorgegebenen Wegpositionen aus *gWegPos* gelesen und durch *perc2pix* in Bildschirmkoordinaten umgerechnet. Die

Ergebnisse werden lokal in *lWegPos* gespeichert. Danach werden alle Einträge in *gZoomSteps* geprüft. Stimmt der Faktor mit dem berechneten Wert von Ziel oder Figur überein, wird ein Vektor zwischen der *Canvas*-Position und dem Wegpunkt auf der gleichen Zoomebene berechnet. Auf diese Weise werden der erste und letzte Bewegungsvektor berechnet. Insgesamt existieren drei Vektoren, die über die Bewegung der Figur von ihrer Startposition zum zentralen Wegpunkt, von dort zum nächsten Wegpunkt und abschließend zum Ziel bestimmen. Diese Aufteilung in drei Abschnitte liegt jeder Bewegung zugrunde. Die drei Vektoren werden in *gMoveVec* gespeichert und bleiben so zwischen den Aufrufen von *bewegePerson* erhalten. Der Faktor *gPixelProAufruf* bestimmt über die Schrittweite und Geschwindigkeit der Figur, indem der berechnete Vektor dadurch geteilt wird. Die beiden anzusteuernden Wegpunktkoordinaten werden in *gTargets* hinterlegt, während *wegindex* die entsprechenden Indexeinträge in der richtigen Reihenfolge speichert. Zusätzlich wird für die Skalierung noch ein Vektor in *gMoveVec* abgelegt. Er gibt die Veränderung der *Canvas*-Größe pro Schritt an und wird auf dem zentralen Pfad angewendet. Sein Betrag ergibt sich aus den Angaben zur Zielgröße in *gTargets*, die eingangs anhand von *zoomFaktor* und *gStartAbmessungen* berechnet wurden, dividiert durch *gPixelProAufruf*.

Die relative Größenveränderung zwischen Start- und Zielabmessungen wird in *zoomFaktor* gespeichert. Sie wird bei der Berechnung der Zielkoordinaten auf dem zentralen Pfad zwischen den beiden Wegpunkten genutzt. Im Gegensatz zur Position der *Canvas*-Elemente müssen diese Werte separat berechnet werden, da sie durch die Größenänderung des bewegten *Canvas* dynamisch sind. Die Berücksichtigung stellt sicher, dass die Figur die Wegpunkte mittig am unteren Ende der jeweiligen Abmessungen ihres *Canvas* trifft. Nachdem die Bewegungsvektoren berechnet wurden, wird *gWegBerechnet* auf *true* gesetzt. Somit erfolgt die Berechnung einmalig zu Beginn der Animation.

Falls das Spiel gerade geladen wird, besteht kein Grund für die Darstellung der Animation. Deshalb wird in diesem Fall, identifiziert durch das *gLoadByCode*-Flag, die Figur direkt an ihr Ziel versetzt und *finishWalking* aufgerufen.

Jeder weitere Aufruf von *bewegePerson* prüft zuerst, ob das Ziel erreicht wurde. Dazu wird die aktuelle Position der Figur mit dem derzeitigen Ziel an *zielErreicht* übergeben. Die Funktion kann mithilfe des Parameters *justDistance* gesteuert werden. Steht er auf *true*, wird nur der Distanzvektor zwischen Ziel und derzeitiger Position bestimmt. Dabei wird auch der aktuelle Wert des z-Index berücksichtigt und unterhalb eines bestimmten, sehr kleinen Abstandes, *true* zurückgegeben. Die andere Berechnungsvariante ermöglicht eine maximale Annährung an das Ziel. Sie berechnet den aktuellen Abstand und den Wert nach dem nächsten Schritt in Richtung des aktuellen Bewegungsvektors. Entfernt sich die Figur im nächsten Schritt wieder vom Ziel, gilt es als erreicht und die Funktion gibt ebenfalls *true* zurück.

Wurde das Ziel noch nicht erreicht, wird das *Canvas*-Element innerhalb von *bewegePerson* in Richtung des aktuellen Wertes von *gMoveVec* verschoben. Während sich die Figur auf dem Pfad zwischen zwei zentralen Wegpunkten befindet, *gAktuellesZiel* also auf eins steht, wird der Inhalt des *Canvas* mit jedem Schritt skaliert. Dazu wird das Element und die Skalierungsfaktoren beider Dimensionen an *skaliereHeld* übergeben. Sie skaliert den Bildinhalt des *Canvas* um die übergeben Werte, ausgehend von den Angaben in *gStartAbmessungen*. Diese werden danach bei jedem Schritt entsprechend angepasst.

Wurde das Ziel bei der vorherigen Überprüfung als erreicht bewertet, wird das *Canvas*-Element der Figur exakt auf den gespeicherten Zielkoordinaten platziert. Dies gleicht kleinere Abweichungen während der Bewegung aus. Auf dem zentralen Pfad wird der Bildinhalt anschließend mit den, zu

Beginn der Wegberechnung erfassten, Werten skaliert. Abschließend wird *gAktuellesZiel* inkrementiert und zeigt so auf das nächste Ziel.

Abschließend wird geprüft, ob das ursprünglich ausgewählte Ziel erreicht wurde. In diesem Fall muss *gAktuellesZiel* den Wert drei angenommen haben. *gStartAbmessungen* wird dann auf die zuvor berechneten Zielwerte aus *gTargets* gesetzt. Dadurch wird, neben der Position, auch die Größe des *Canvas* korrigiert. Abschließend wird *finishWalking* aufgerufen.
Für den Fall, dass das Ziel noch nicht erreicht wurde, wird zunächst die Laufrichtung durch *determineWalkingDirection* bestimmt. Die Funktion erzeugt aus der aktuellen *Canvas*-Position eine Instanz der Klasse *lastValidInformation* und vergleicht sie mit der letzten bekannten Position in *gLastValidPositionData*. Deren Wert wird beim Aufruf der Seite automatisch auf den Nullpunkt gesetzt. Aus der Differenz zwischen beiden Positionen ermittelt die Funktion die Bewegungsrichtung und aktualisiert *gLastValidPositionData* mit der aktuellen Position. Falls keine korrekte Richtung ermittelt werden konnte, wird automatisch *gLastDirection* gesetzt, die zuletzt ermittelte Laufrichtung. Der Sonderfall eines Außeneinsatzes im Weltraum, wie er in der Handlung vorkommt, wird hier ebenfalls durch die Auswertung des *gSpace*-Flags berücksichtigt. Das Ergebnis der Auswertung wird an *switchWalkingAnimation* übergeben.

Zum Abschluss der Bewegung wird in *finishWalking* die endgültige Blickrichtung bestimmt und angewendet. Darüber hinaus wird der zweite Fall, eine Bewegung der Figur in die Tiefe der Szene, hier behandelt. Das *Canvas*-Element wird entsprechend verkleinert. Anschließend wird *gAnimationTimer* nach dem entsprechenden Eintrag durchsucht und die Zielgröße der Figurenanimation angepasst, indem die Werte aus *gStartAbmessungen* übernommen werden. Hierbei wird auch die Transformationsmatrix des *Canvas* zurückgesetzt, die während der Skalierung verändert wurde.

Abschließend werden alle globalen Variablen zurück, *gEventHandlerBusy* auf *false* und das Dialogfeld wieder sichtbar gesetzt. Sofern das Spiel nicht per Code geladen wird, erfolgt auch ein Aufruf von *finishEventHandling*.

Dialogausgabe

Die Ausgabe der Dialoge erfolgt mithilfe der Funktionen in *dialogControl.js*. Beim Laden der Seite wird jedoch bereits das Objekt *gTalk* angelegt. Es speichert die Parameter zur Textausgabe, wie zum Beispiel Schriftart, Zeilenabstand und Schriftfarbe. Darüber hinaus enthält es auch die Standardposition der einzelnen Elemente der Dialogbox.

Die Funktion *advanceDialogStep* wird bei der Verarbeitung eines Klicks in *clickEventHandler.js* aufgerufen. Ihr werden die IDs des geklickten Bildes übergeben, einmal in Form der Bild-ID, einmal inklusive aller steuernder Flags. Nachdem überprüft wurde, ob das geklickte Objekt im aktuellen Rätselschritt als Auslöser gekennzeichnet ist, wird mit der Verarbeitung fortgefahren. Zunächst wird die Funktion *fetchDialogIDs* aufgerufen. Sie gibt in Abhängigkeit von *gDialogCounter* entweder den Inhalt von *gDeprecatedDialogIDs* oder *gDialogIDs* zurück. Die Variable gibt dabei den Index des aktuellen Dialogs im Array an. Da es unter bestimmten Umständen vorkommen kann, dass das zurückgegeben Array leer ist, wird es in diesem Fall durch *gBackupOfDialogs* ersetzt. Damit stehen dann alle eingelesen Dialoge zur Verfügung.

Die ID des angeklickten Bildes wird als Vergleich für den entsprechenden Eintrag in *gImageToObjectSceneReferrer* genutzt. Es enthält unter anderem die dem Objekt zugewiesenen Dialoge. Diese werden anschließend durchlaufen und jeder durch *testIfSubDialog* geprüft. Die Funktion nutzt *patternTest* und *idIsBlacklisted* sowie Flag *isInitialized* in *gTalk* als Indikatoren für gültige Einträge.

patternTest prüft dabei das Vorhandensein von maximal drei Zahlenwerten im Dialogbezeichner. Sie kennzeichnen die Abfolge der Dialoge in hierarchischer Reihenfolge und bestimmen Dialoggruppen.

idIsBlacklisted prüft das Array *gSubDialogBlacklist* auf einen Eintrag mit der ID des gegebenen Dialogs. Es enthält alle Dialog-IDs, die bei einem bestimmten Spielfortschritt, bestimmt durch *gDialogCounter*, nicht angezeigt werden sollen. Das Array wird in *blackListLeftOverSubDialogues* mit allen Dialogen gefüllt, die zu einer Gruppe gehören. Dazu prüft die Funktion, ob alle Gruppenmitglieder bis zum Gegebenen angezeigt wurden. Sie sollten in *gSubDialogBlacklist* geführt werden und werden, wenn nötig, in Form einer *BlacklistIDObject* –Instanz hinzugefügt. Sie enthält die Dialog-ID und den aktuellen Wert von *gDialogCounter*.
Diese Maßnahme verhindert die verspätete Anzeige von Dialogen einer Gruppe, die zuvor nicht dargestellt wurden.

Entsprechend dieser Indikatoren gibt die Funktion *testIfSubDialog true* zurück wenn es sich um einen Dialog innerhalb einer Dialoggruppe handelt. Andernfalls wird der übergebene Dialog in das Array *gSubDialogBlacklist* aufgenommen und alle zugehörigen Gruppendialoge erfasst. Ihre Anzahl wird in *gSubDialogCount* gespeichert. Danach werden alle Dialoge der Gruppe durchlaufen. Der Index des, der Funktion übergebenen, Dialogs wird in *gSubDialogOffset* erfasst. Handelt es sich dabei um den letzten Dialog einer Gruppe wird wiederum *true* zurückgegeben und dieser Dialog in *blackListLeftOverSubDialogues* verarbeitet.

Durch diese umfangreichen Prüfungen wird *gIncreaseDialogStep* so gesetzt, dass nur die letzten Dialoge einer Gruppe ein Inkrementieren von *gDialogCounter* auslösen. Dadurch werden Dialoggruppen möglichst vollständig angezeigt.

Die Funktion *advanceDialogStep* führt danach eine Prüfung auf Vorhandensein des ermittelten Gruppendialogs innerhalb von *gDeprecatedDialogues* beziehungsweise *gDialoge* durch. Sollte er sich nicht in diesen Variablen befinden, muss von einem generellen Programmfehler ausgegangen werden. Die Überprüfung der, für das geklickte Szenenobjekt in *szenen.xml* hinterlegten, Dialoge wird fortgesetzt und jeder Eintrag, welcher nicht dem ermittelten Gruppendialog entspricht, wird übersprungen. Die Prüfung erfolgt dabei anhand des Index, zusammengesetzt aus der Summe von *gDialogCounter* und *gSubDialogOffset*, innerhalb von *dialogIDs*.

An diesem Punkt sind alle Voraussetzungen zur Darstellung des Textes erfüllt. Dementsprechend wird in *gTalk* das Attribut *dialog_id* auf den ermittelten Dialog gesetzt und *dialog_zeichneDialog* aufgerufen. Danach wird, abhängig von *gIncreaseDialogStep*, gDialogCounter um eins und zusätzlich um die Anzahl von Gruppendialogen erhöht. Abschließend werden *gSubDialogCount* und *gSubDialogOffset* zurückgesetzt.

In der Funktion dialog_zeichneDialog werden zunächst wieder die Dialog-IDs mit *fetchDialogIDs* erfasst. Der anzuzeigende Dialog wird nun dahingehend überprüft, ob er eine Ausnahmebehandlung auslöst. Hierzu wird die Eigenschaft invoke_scene_*exception* ausgewertet und dementsprechend *triggerException* aufgerufen. Falls diese Aktion einen anderen Dialog erzwingt, wird dessen ID aus *gDialogToForce* gelesen und in *gTalk.dialog_id* gespeichert. Die Erkennung erfolgt durch das Flag *gForceOtherDialog*, welches in den Ausnahmebehandlungen in *sceneException.js* gesetzt wird.

Anschließend werden die Eigenschaften *currentDialog* und *SatzMax* von *gTalk* gesetzt. Sie repräsentieren den anzuzeigenden Dialog und die Anzahl der in ihm enthaltenen Sätze. Zusätzlich wird *isInitialized* hier auf *true* gesetzt.

Danach erfolgt eine Überprüfung, ob der zu zeichnende Dialog das Rätsel vorantreibt. In diesem Fall wird *advanceQuizStep* mit dem Flag „CalledByDialogue“ aufgerufen. Die Prüfung erfolgt anhand der

Flags *trigger_quizstep* und *enable_at_start* des Dialogs. Sie werden direkt aus *dialoge.xml* gelesen und entscheiden über die erzwungene Anzeige des Dialogs zu Beginn einer Szene und seine Eigenschaft das Rätsel voran zu treiben.

Die Funktion liest danach den aktuellen Satz innerhalb des Dialogs aus. Sein Inhalt wird in *Text* gespeichert und *swapProxiesWithNames* übergeben. Die Funktion ersetzt die Platzhalter für die Namen der Hauptfiguren, gespeichert in *gP1Proxy* und *gP2Proxy*, durch die Werte in *gP2Name* und *gP2Name*. Alternativ werden die Sicherungswerte aus *gFallbackNameP1* und *gFallbackNameP2* verwendet, falls keine Namen definiert wurden.

Im weiteren Verlauf wird dem Satz ein Trennzeichen am Ende hinzugefügt, falls es sich nicht um den letzten Satz im Dialog handelt. Danach werden die Bildschirmabmessungen und die Größe der Dialogbox erfasst. Die Funktion *getScaledDimensions* in *sceneException.js* berechnet dabei die Größe eines *Canvas* im aktuellen Fenster. Mit diesen Parametern werden die Positionsangaben aller Dialogboxelemente in *gTalk* berechnet. Dazu zählen das Bild des Sprechers sowie des Textes.

Die Schriftgröße wird durch die Funktion *changeFontSize* ebenfalls an die Bildabmessungen angepasst. Ihr wird die aktuelle Bildhöhe, geteilt durch *gPercentageFontSize*, übergeben. Der Wert bestimmt die relative Schriftgröße im Verhältnis zu den Abmessungen. In der Funktion wird das Array *font_style* in *gTalk* bearbeitet. Es enthält CSS-ähnliche Angaben von denen nur die Schriftgröße ausgetauscht wird. Zusätzlich wird *line_distance* entsprechend angepasst.

In dialog_*zeichneDialog* wird nun *pixSize* bestimmt. Dessen Wert richtet sich nach den berechneten Abmessungen und Koordinaten und bestimmt später über die Trennung der Sätze in *dialog_SatzZeilenBruch*. Zunächst wird das *Canvas* der Dialogbox erfasst und geleert. Das Hintergrundbild wird danach erneut gezeichnet und dann durch das Bild des Sprechers überdeckt. Hierzu werden die vorher berechneten Abmessungen, *ProtImgWidth* und *ProtImgHeight*, und Positionen, *ProtImgXPos* und *ProtImgYPos*, verwendet. Schriftstil und –farbe werden aus den *gTalk* Attributen *font_color* und *font_style* übernommen.

Der Satzinhalt in *Text* wird durch *dialog_SatzZeilenBruch* in einzelne Zeilen, die in die Dialogbox passen, aufgeteilt. Dazu werden zunächst alle Kommata im Text durch das Flag „#KOMMA#“ ersetzt, um eine falsche Interpretation als Arrayseparator zu verhindern. Danach wird der Text so lange in Abschnitte zerlegt bis dies keine Veränderung mehr ergibt oder alle Abschnitte ermittelt wurden. Die Trennung erfolgt dabei an der Position des letzten Leerzeichens innerhalb eines Textabschnitts dessen Länge *pixelSize* entspricht.

Nach dieser Aufteilung wird der Satz zeilenweise in das *Canvas* geschrieben, wobei die Kommata wieder eingefügt werden. Abschließend werden *SatzCounter* in *gTalk* inkrementiert und, falls es sich um den letzten Satz im Dialog handelt, alle beteiligten Variablen wieder zurückgesetzt.

Zur Darstellung erzwungener Dialoge zu Beginn einer Szene wird die Funktion *forceDialog* genutzt. Sie nutzt stets den ersten angegebenen Dialog aus dialoge.xml. Nach dem Aufruf von *dialog_zeichneDialog* wird *gDialogCounter* inkrementiert, falls *isInitialized* in *gTalk* auf *false* steht. Dies ist am Ende eines Dialoges der Fall.

Rätselkontrolle

Der Fortschritt des Spielgeschehens wird hauptsächlich von *quizControl.js* kontrolliert. Die Datei beinhaltet dazu die Methode *advanceQuizStep*. Durch die Übergabe der Flags in *raetsel_ausloeser* aus der *szenen.xml* wird hier geprüft, ob das geklickte Objekt um Rätsel beiträgt. Alternativ kann ein Fortschritt durch die Übergabe von „CalledByDialogue“ erzwungen werden. Bei einem positiven

Ergebnis der Auswertung wird zunächst *gCurrentQuizstep* inkrementiert. Die Variable repräsentiert den aktuellen Fortschritt im Rätsel und dient zum Beispiel als Index für die Flag-Arrays.

Zur Anpassung der Szene wird danach *applyCSSClass* aufgerufen. Hier werden alle Objekte der aktuellen Szene durchlaufen und die ihnen zugeordneten Flags *raetsel_sichtbar* und *klickbar* ausgewertet. Die Anpassung erfolgt durch eine Neuzuweisung der CSS-Klassen *quiz_shown*, *quiz_hidden* und *clickable*. Während die ersten beiden Klassen die Elemente lediglich verstecken oder wieder anzeigen, fügt die Klasse *clickable* ihm eine Animation hinzu, welche durch Berührung mit dem Mauszeiger ausgelöst wird. Dies hebt das Element als interaktiv hervor. Jeder Eintrag der *szenen.xml* ohne ausreichend viele Einträge in den Flag-Arrays wird an dieser Stelle ausgeblendet, sobald *gCurrentQuizstep* über die Anzahl der hinterlegten Einträge hinaus wächst.

Innerhalb von *advanceQuizStep* wird nun *checkQuizfinished* aufgerufen. Die Funktion prüft, ob das Szenenrätsel gelöst wurde und leitet das Laden der nächsten Szene ein. Dabei wird zwischen zwei Zuständen unterschieden.

Im vorletzten Rätselschritt, erkannt durch den Vergleich von *gQuizsteps*, der Summe aller Schritte, und *gCurrentQuizstep*, wird der Inhalt der nächsten Szene geladen. Durch die Berücksichtigung von *gSceneHasBeenLoad* wird dieser Abschnitt in jedem Fall nur einmal aufgerufen. Die Variable zeigt den in Auftrag gegebenen Ladevorgang an und wird beim Szenenwechsel zurückgesetzt. Zunächst werden jedoch alle Bilder in *gBilder* nach *gDeprecatedImages* verschoben, *gUseDeprecatedImages* auf *true* gesetzt und zur Vorbereitung des Ladevorgangs *gBilder* geleert. *gcurrent_scene_counter*, welche die aktuelle Szenennummer enthält, wird danach inkrementiert, um auf die nächste Szene zu zeigen. Abschließend wird der Ladevorgang durch *ladeBilder* und *ladeDialoge* initiiert. *gSceneHasBeenLoad* und *gUseDeprecated* werden auf *true* umgeschaltet.

Im letzten Rätselschritt erfolgt der Szenenwechsel. Dazu wird zunächst ein neues *Canvas*-Element erzeugt, welches die Übergangsanimation beinhaltet. Derzeit ist hier ein einfacher Überblendungseffekt vorgesehen. Die Eigenschaften dieses Effektes werden ausgelesen und in *frametime* und *framecount* gespeichert. Dadurch ist es möglich, *advanceNextScene* durch Zeitgeber genau nach der halben Dauer der Übergansanimation aufzurufen. Dadurch wird der eigentliche Szenenwechsel von der Animation verdeckt. Nach einem Durchlauf wird sie wiederum angehalten und das entsprechende C*anvas* entfernt.

Der Aufruf von *advanceNextScene* dient größtenteils dem Zurücksetzen der Variablen für die neue Szene. Darüber hinaus werden alle laufenden Animationen angehalten und die Reste der alten Szene entfernt. Zum Schluss wird *gdisplay_next_scene* auf *true* gesetzt und über *waitforparser* letztendlich *sceneParser.js* aufgerufen.

Codegenerator

Jede relevante Aktion in *clickEventHandler.js* bedingt den Aufruf von *verschluesseln* in *codeGenerator.js* sowie ein Inkrementieren von *gCodegeneratorIndex*. Die Variable zeigt auf einen, dem geklickten Element zugehörigen, Eintrag in *gCodegeneratorArray*.

In der Funktion werden die aktuelle Szenennummer und der genannte Index zu einem eindeutigen Schlüssel zum Laden der Szene verarbeitet. Dazu wird zunächst zufällig ein Buchstabe gewählt, welcher als Trennzeichen fungiert. Danach werden die beiden Zahlen mithilfe von *gNumberToNumeral* in Strings kodiert. Im Array sind dazu entsprechende Werte für alle Zahlen bis 59 hinterlegt. Die Ergebnisse der Ersetzung werden zusammen mit dem Trennzeichen zu einem Wort zusammengesetzt.

Dieses Wort wird nun durch einen Algorithmus verschlüsselt. Zunächst wird der American Standard Code for Information Interchange (ASCII) jedes Zeichens ermittelt und dessen Index um 13 Stellen verschoben. Wurde dabei der gültige Bereich des kleingeschriebenen Alphabets überschritten, erfolgt eine Anpassung, sodass das Ergebnis wiederum innerhalb liegt. Danach wird jedes Zeichen im Schlüssel in einen Großbuchstaben umgewandelt, wenn es entweder innerhalb der zweiten Hälfte des Alphabets liegt oder der Schleifenindex an seiner Position einen ungeraden Wert annimmt.

Abschließend wird dem Schlüssel das Trennzeichen vorangestellt, welches zuvor zufallsgesteuert in einen Großbuchstaben umgewandelt wurde. Die Funktion gibt den erzeugten Schlüssel am Ende zurück.

Zur Entschlüsselung wird wiederum *entschluesseln* aufgerufen. Die Funktion führt denselben Algorithmus in umgekehrter Reihenfolge aus und gibt die Werte für *gcurrent_scene_counter* und *gCodegeneratorIndex* zurück.

Nach der Eingabe eines Schlüssels im Spielmenü wird *spiel_fortsetzen* aufgerufen und entschlüsselt zunächst die Eingabe. Danach werden *ladeBilder* und *ladeDialoge* aufgerufen wodurch anschließend auch *sceneParser.js* durchlaufen wird. Die Funktion setzt ebenfalls *gLoadByCode* und bewirkt dadurch in *getSceneInformation* den Aufruf von *advanceSceneToLastSavestate* nach dessen Beendigung *gLoadByCode* wieder zurückgesetzt wurde.

Die Funktion simuliert die Eingabe aller nötigen Aktionen bis zum Erreichen des durch den Schlüssel definierten Zustandes. Dabei werden alle für die Szene hinterlegten Objekte in *gCodegeneratorArray* nacheinander aufgerufen. Jeder Aufruf wird verarbeitet, als wäre er durch einen Klick des Spielers hervorgerufen worden. Der einzige Unterschied besteht in der Darstellung der Laufanimation. Der aktuelle Wert für *gPixelProAufruf* wird gesichert und nach dem Ladevorgang wiederhergestellt. Währenddessen steht er auf eins und bewirkt so eine Animation mit maximaler Geschwindigkeit, indem die Figur unmittelbar auf ihr Ziel gesetzt wird.

Ausnahmebehandlung